IN-60

2421

114P

NASA Contractor Report 194884

# USING FORMAL SPECIFICATION IN THE GUIDANCE AND CONTROL SOFTWARE (GCS) EXPERIMENT

Doug Weber and Damir Jamsek
*Odyssey Research Associates, Ithaca, NY*

Contract NAS1-1897

April, 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, VA 23681-0001

# Using Formal Specification in the Guidance and Control Software (GCS) Experiment

# Final Report

**Formal Design and Verification Technology for Life Critical Systems**
**NASA Contract No. NAS1-18972**
**ORA Task Assignment 7**

Prepared for:
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Prepared by:
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326

4 April 1994

# Contents

# Chapter 1

# Overview

Under the NASA/Langley project titled "Formal Design and Verification Technology for Life Critical Systems", ORA Corporation is researching better ways to develop reliable software and hardware. This document is the Final Report for task 7 of that project. Task 7 is specifically titled "Using Formal Specification in the Guidance and Control Software Experiment".

## 1.1 Introduction

### 1.1.1 The Guidance and Control Software

NASA's Guidance and Control Software (GCS) Experiment is a study of methods for developing flight control software. The goal of the study is to understand the sources of software defects, and to find ways to detect and prevent them. In the experiment, engineers implement GCS several times using various methods; the effect of different methods can then be compared. [4].

GCS is software to control the descent of a spacecraft onto a planet's surface. The spacecraft is a simplification of the 1970's Viking Mars lander. By simplifying the lander, the GCS requirements, design, and code are all shortened, making it possible to implement GCS several times within the budget of the experiment.

Although simplified, GCS still presents many of the features that characterize control software. These features will be discussed in the course of the report. Control software differs in important ways from other kinds of software, particularly in the kinds of requirements placed on it. These differences affect the methods engineers use to develop software; the report will discuss these differences.

### 1.1.2 ORA's Task

This task has two parts:

1. apply formal methods of software engineering to GCS;
2. understand how formal methods could be incorporated into a software engineering process for flight control systems.

Both parts of the task are open-ended. There is more than one approach to the use of formal methods, and within each approach, even partial use of formality can add to confidence in the software developed. We have begun to apply formal methods by formally specifying the software requirements for GCS. In the next section we will explain what this statement means as we introduce the terminology of formal methods.

Defining a good process for developing software is a topic of great current interest. The aerospace community has already created a standard, DO-178B [3], for software engineering process. This standard does not determine a specific process, but rather constrains the possible processes that may be chosen when developing flight control software. Software engineers then choose a specific process that satisfies the constraints.

The Do178B standard lists formal methods as an acceptable atlernate means of compliance. A subgoal of our task is to propose a software development process, consistent with DO-178B, in which formal methods are used.

## 1.2 Formal Methods

*Formal methods* bring the precision of mathematical notation, theory, and reasoning to bear on the problem of understanding the behavior of computer systems. In particular, a *formal specification* is a mathematical constraint on a system's intended behavior, and a *formal verification* is a mathematical proof that the algorithms used to implement the system meet the specified constraint. This leads to demonstrably correct systems, where the assumptions and reasoning on which correctness depends are made explicit.

Formal methods are intended to augment the informal methods conventionally used in system development. The conventional methods of testing and debugging, when applied to very complicated systems, have often permitted incorrect, unexpected, or undependable behavior due to conceptual flaws in design or implementation. This happens because the informal methods analyze system behavior for particular inputs only. In contrast, the method of formal verification can apply to all possible inputs, and so one can analyze at once all the system's possible behaviors.

Formal methods might be applied to any phase of system development, including requirements analysis, design, implementation of both hardware and software, choice of test cases, and so forth. In this task we have concentrated on formally specifying the highest level of software requirements in GCS.

Formal methods can be carried out manually or with the support of automated tools. Tool support is especially desirable for managing the complexity of verification proofs. However, in this task we did not carry out any proofs, and our reliance on tools has been minimal.

## 1.3 Outline of the Report

In Chapter 2, we describe the lander to be controlled by GCS. The requirements on GCS are stated in general terms. Our detailed, formal specification of GCS requirements appears in Appendix A.

In Chapter 3, we discuss the process of requirements analysis for control software such as GCS. We show that our requirements specification could be improved if more information were available about the system engineering decisions made in the lander's design. Specific criticisms of an earlier, informal specification of GCS requirements are listed in Appendix B.

Chapter 4 relates formal methods to the DO-178B framework. Finally, Chapter 5 draws some conclusions about this work.

# Chapter 2

# The Guidance and Control Software

To understand GCS we must understand the lander it is designed to control. In this chapter we describe the lander. We also describe some key ideas underlying the control of the lander. The detailed software requirements on GCS can be found by reading the formal specification of GCS in Appendix A.

The lander is designed to descend through a planetary atmosphere and land safely. The descent passes through several stages.

- The lander falls, uncontrolled except by parachute, until a predefined altitude is reached.
- Once that altitude is reached, the engines begin warming up. Active control begins.
- Once the engines are hot, the parachute is released. Active control continues.
- After a second predefined altitude near the ground is reached, or the ground is directly sensed, the engines are shut off and the lander drops to the ground.

The landing is considered safe if the speed is small enough when the engines are shut off.

The descent has additional properties:

- The altitude measurement depends on GCS, so GCS processing must be started before reaching the altitude at which the engines are turned on.

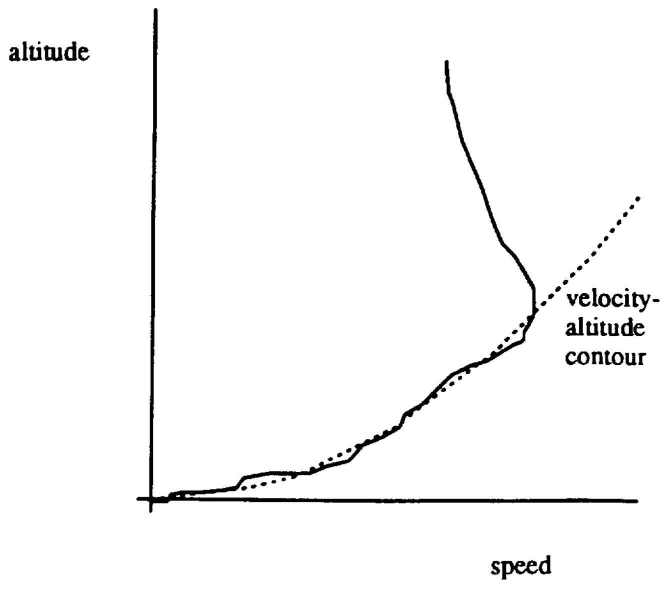


Figure 2.1: A controlled descent

- After releasing the parachute, active control of the lander aims to maintain a predetermined relation between the lander's speed and its altitude. This relation is called the "velocity-altitude contour". An example contour is plotted in Figure 2.1 along with a trajectory of the lander showing speed and altitude.
- The active control differs at different stages of the descent.
  - During engine warm-up, simplified control laws are used and less thrust is applied than after the engines are hot.
  - Once the engines are hot, the lander's attitude is controlled, but the lander is permitted to accelerate until its speed first becomes greater than allowed by the velocity-altitude contour. This event is called "crossing" the contour.
  - Once the contour is crossed, both speed and attitude are controlled.

To describe the lander's motion in more detail, we use a right-handed, rectangular coordinate system fixed with respect to the lander. The positive x-axis of this coordinate system points downward along the lander's vertical axis. *Roll* is rotation around the x-axis; *pitch* is rotation around the y-axis; *yaw* is rotation around the z-axis.

The lander interacts with its environment using a variety of sensors and actuators, and in addition it uses a radio transmitter to send data back to an orbiting platform. The lander has the following actuators:

- Three **axial engines** create thrust in the negative x direction. The thrust from each engine is separately controllable. To slow the lander down, thrust is applied equally from all axial engines. To alter pitch or yaw, torque is supplied by differences in the thrust from the three engines.

- Three opposing pairs of **roll engines** are mounted on the sides of the lander, perpendicular to the x-axis. These roll engines provide torque around that axis; the torque is used to control roll.
- A **chute release** mechanism exists for releasing the parachute.

The lander has the following sensors:

- An **altimeter radar** measures the lander's altitude by timing the echoes of radar pulses bounced off the planet's surface.
- Each of three **accelerometers** measures the lander's acceleration along one coordinate axis.
- Each of three **gyroscopes** measures the lander's rate of rotation around a coordinate axis.
- Each of four Doppler **touch-down landing radars** measures the lander's speed in a single direction. The speed measurements are combined into a measure of the velocity.
- Two **temperature sensors**, one solid-state, the other a matched pair of thermocouples, yield a temperature measurement that is used to correct data from some of the other sensors.
- A **touch-down sensor** attached to the end of a rod under the lander, detects when the ground is touched.

Although the lander is a simplified version of the real Viking system, the following factors are realistic and make GCS's control task interesting:

- Data from a variety of sensors must be processed.
- Sensor data may be missing or less precise than needed for control, and some of the lander's physical degrees of freedom cannot be measured directly, so GCS processing must fuse the sensor data into a precise, coherent model of the lander's dynamics.
- Control laws must be specified for a variety of the lander's degrees of freedom.
- Some degrees of freedom are affected by several actuators, so the actuators must be coordinated.

The control laws for the actuators have these goals:

- The roll angle (defined as the integral of the roll rate), is maintained constant. Controlling roll in this way is desirable for stability so that the control of the other degrees of freedom does not need to compensate for roll rotation.
- The pitch and yaw angles are controlled so that the lander's x-axis points along the velocity vector. This ensures that axial engine thrust decreases speed.
- The descent follows the velocity-altitude contour. The contour is (presumably) chosen to minimize the use of fuel and meet other engineering constraints.

The formal requirements specification in Appendix A gives much more detail about GCS. The appendix can be read and understood after reading this chapter. The formal specification is based on a previous informal requirements specification for GCS [7]. That informal specification document contains some helpful illustrations of the lander and some background references.

The next chapter discusses general problems of specifying requirements, both formally and informally. GCS is used as an example.

# Chapter 3

# Formal Specification of Control Software

Control software differs from other kinds of software. The difference affects the kinds of requirements placed on the software, and the process used to decide on those requirements.

Because GCS is control software, we needed to understand this difference before writing a formal specification of GCS requirements. This chapter presents our analysis.

Control software is special because the purpose underlying it cannot be expressed without referring to the state of the physical world. The purpose behind GCS, for example, is to ensure that the lander lands, but does not crash. This purpose has nothing to do with software, and everything to do with a non-digital, analog reality.

In contrast, the purpose of other kinds of software can often be expressed solely in terms of digital input and output, or the state of digital devices. The purpose of a screen editor, for example, is to make the state of the screen a function of the state of the disk and a history of user commands. In this case, the software designer can and should make assumptions about the non-digital world that allow that world to be ignored.

Because the control software cannot ignore the external world, the process of formulating software requirements must be tightly coupled to the engineering decisions about the system being controlled. We will address the question: What form should this coupling take?

## 3.1 Specifying Software Requirements

A requirements analyst should strive to identify and express requirements that have the following two properties:

1. **completeness:** every requirement should be stated unambiguously.
2. **clarity:** only the requirements should be stated, and these should be kept as simple and as understandable as possible.

Obviously the completeness property is essential for developing reliable software. Without a complete specification, a programmer must resolve ambiguities in some way, perhaps by guessing, in order to implement the software. Guessing is not reliable.

The need for clarity is less obvious (judging, at least, by the commonness of massive, redundant, and overly detailed software requirements documents). Clarity is important for the following reasons:

- Without a clear specification, a programmer is constantly wondering what the purpose of the software is, and how to extract specific requirements from a lot of detail.
- An unclear specification makes it harder for the programmer to see when the specification itself is in error.
- A requirements specification that is unclear because it contains too much detail is in effect a design document. It is a mistake to mislabel design decisions as requirements because
  - the programmer's freedom to choose the best design is restricted;
  - when the true requirements are changed, modifying a statement of requirements that includes part of the design is harder to do consistently.

Writing a clear requirements specification depends on abstraction. If possible, the purpose of the software should be stated as the abstract requirement. In the example of the screen editor again: "the state of the screen is a function, $f$, of the state of the disk and the history of user commands". More detailed requirements can be shown to imply the abstract requirement. In the screen editor example, the unspecified function, $f$, could be *refined* by giving some of its properties. The process of refining the abstract requirement into supporting requirements stops when the programmer has enough information to proceed.

### 3.1.1 Formal Specification

Formal specification is often touted as a method for writing better requirements. This claim has merit. Certainly a formal requirements specification is more likely to be complete than an informal one, because the activity of formalization forces ambiguities to be addressed.

On the other hand, formal specification does not necessarily help clarity: writing a overspecification is just as easy in a formal language as in an informal one. Also, specifiers sometimes find expressing abstract requirements formally to be a problem because of limitations of a particular formal language.

Most of our discussion will apply both to formal and informal specifications.

## 3.2 Control Software Specification

When specifying control software, the need for both completeness and clarity creates a dilemma: should the software requirements be abstract, and in the process describe the physical system being controlled and the environment of that system? Or should the software requirements be concrete, separating the concerns of the software requirements analyst from those of the system engineer and physicist?

The clearest software specification would be an abstract statement of the properties of the controlled system. In the GCS example, "the lander lands without crashing". However, this specification is not directly a requirement on *software*; it is quite possible to satisfy it without any software at all. To produce a complete specification we must refine this abstract requirement into a collection of concrete requirements the programmer can work with, as diagrammed in Figure 3.1.

The refinement process, however, is essentially the work of the system engineers who designed the lander. Reproducing this engineering as part of the software requirements, using the language of the software requirements, and validating the reasoning behind each step, could be quite tedious and verbose. Might we just skip this formality and simply use the collection of concrete requirements as the software specification?

At least one example in the formal methods literature [2] carries out a control system refinement as part of formally verifying the abstract system requirement. The example is the control of a vehicle buffeted by crosswind: the control software steers a straight course even though the environment is unpredictable. This example verifies the design of the software by proving an abstract requirement about the *system*, including the vehicle, the wind, etc. While the example shows that this kind of refinement and verification is possible, the example is very simple, and it is not clear that the approach it advocates would scale up to larger control systems.

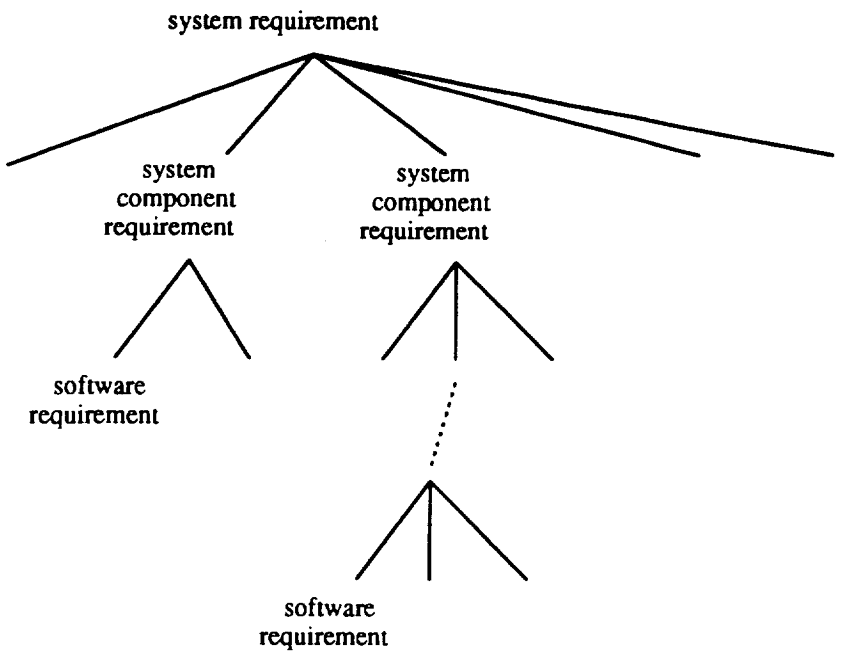


Figure 3.1: System requirements analysis

We contend that this approach is not good, for the following reasons:

- The complexity of the system engineering in a control system bears little relation to the complexity of the software engineering needed to implement the control. At one extreme, a simple control system might contain a large amount of code; at the other extreme, a complicated control system might contain only a small amount of code. In the latter case, demanding that the software requirements include parts of the system engineering would greatly increase the size of the software task.

- The system engineer justifies his work using different kinds of reasoning than the software engineer. In particular, system engineering typically uses probabilistic arguments and continuous mathematics, both of which are unusual in the practice of software engineering. These different kinds of reasoning should be separated.

- For complicated environments, simulation often replaces rigorous reasoning as the primary analysis tool. When simulation is used, the requirements analyst does not prove that the abstract system requirements are met, but rather offers only supporting evidence for them. As before, this kind of reasoning should not be mixed with the reasoning about software requirements.

- Generally the system engineering disciplines are better understood than the software engineering activities, and so should not be included as part of software engineering. The software engineering task should not be broadened to include system engineering as well; it is hard enough by itself. Software engineers should consider using formal methods to improve *software*, not to try to formalize fields of engineering that are relatively well understood.

An approach that separates system engineering concerns from software engineering would be better. This approach recognizes that different people, with different specialties and different tools will be responsible for different kinds of engineering. The result is a hierarchical decomposition of the requirements analysis, as in Figure 3.1.

Unfortunately, the approach we advocate means that the software requirements for control software will tend to be relatively concrete. They will not state the abstract goal of the software. If taken in isolation, without the requirements analysis for the system, they will appear *ad hoc*.

We advocate an approach in which the software requirements are as abstract as possible given the condition that they do not describe the physical system being controlled. In the next section we use the example of GCS to see what this approach means.

### 3.2.1 GCS Requirements Analysis

Let us use the GCS lander as an example to see the kinds of engineering decisions that must be made in a specific control system before the software requirements can be stated. In later sections, we will extract from this requirements analysis those features that will be shared by most or all control systems.

Note that this refinement could be done in more than one way. No documents describing the lander's requirements analysis were available to us, so some of the refinement steps are speculative.

#### Level 0

At the most abstract level, we simply require that the lander lands without crashing.

This requirement cannot be guaranteed in its current form, because nothing has been stated about the lander's environment. We must make assumptions about the environment before the system engineering can begin. For example, we assume that the winds encountered by the lander do not blow too strongly. Many other assumptions about the environment are needed.

Even assumptions in this form cannot be certain. Generally we can only claim that the environment violates our assumptions with some acceptably small probability. We may need to assume a probability distribution for some environmental factors.

### Level 1

The first step in refining the level 0 requirement might be as follows. Decide on some dynamical properties of the lander, such as its mass and fuel supply. Find a set of trajectories of the lander that satisfy the level 0 requirement without running out of fuel.

A *trajectory* is a path in an abstract space of *configurations*, where each configuration tells everything we need to know about the lander's dynamical state. For example, the configuration will include the lander's altitude and velocity. It may also include other physical quantities.

In the GCS experiment, the set of trajectories is defined by the velocity-altitude contour, which is a relation between the lander's speed and altitude. A trajectory is in the set if it is acceptably close to the velocity-altitude contour. The meaning of "acceptably close" is not defined in the GCS documentation; if written out, however, it would be a definition of distance in the configuration space, with either an upper bound on distance, or a constraint on the distribution of distances.

Figure 2.1 shows one acceptable trajectory for GCS.

The velocity-altitude contour might be gotten in any of several ways. Perhaps it is derived analytically, possibly using the variational calculus and minimizing the lander's fuel consumption. Perhaps it is found to be a good choice by using simulation. Whatever the method, once we define the contour and acceptable deviations from it, we can (at least in principle) prove that the level 0 requirement is met. The level 1 requirements are sufficient conditions.

A proof of the level 0 requirement depends on the level 0 assumptions about wind, etc. but also on new requirements on the lander. For example, we must make new requirements about the efficiency of the lander's engines for turning fuel into thrust. These requirements constrain the engines' design.

### Level 2

We next refine the level 1 requirement into more specific level 2 requirements on the lander's design. The level 2 requirements must be sufficient to guarantee the ones at level 1.

Level 2 includes the control laws to be enforced. These laws are a mapping from the lander's configuration to an impulse needed to control the lander. An *impulse* may be literally a change in momentum (this is the technical meaning of the term) or more generally some effect produced by the lander's actuators.

Level 2 also includes requirements on the precision with which the control must be implemented. It also includes an upper bound on delays between the time of external disturbances and the response time of the control. Each of these requirements constrains the design of the system and its software.

One way to show that the level 2 requirements imply those at level 1 is to construct an inductive proof. To do this, we show that the lander starts on an acceptable trajectory. We also show that if the lander is following an acceptable trajectory now, then the level 2 requirements, plus the level 0 assumptions about the environment, plus the physics of the lander, imply that the lander will continue to follow an acceptable trajectory for some time step in the future. The induction proves that the lander always follows an acceptable trajectory. The detailed proof might be quite difficult, the time step might be dependent on the configuration, and it would involve reasoning about probabilities, but in principle it could be carried out.

### Level 3

The level 2 requirements can be refined into level 3 requirements, some of which are constraints directly on GCS. At this level, we find the requirements that a programmer needs to begin the task of implementing GCS.

GCS processing is to be implemented as a sequence of *frames*, each of which processes new sensor data, estimates the lander's current configuration, and determines new outputs to the actuators. We will call the estimate of the configuration a *model*. The single step, control, at level 2 has now been refined into three steps: sense, model, act.

The level 2 requirement on response time is a constraint on the length of each frame. Level 3 requirements on the software processing time for one frame, together with specific requirements on the response time of sensors and actuators, plus assumptions about allowable concurrency between hardware and software, will imply the level 2 requirement.

The level 2 requirement that control laws be implemented is implied by level 3 requirements that the sensor and actuator processing be accurate, and that the model be accurate.

The level 2 requirement on the precision of the lander's control is implied by a combination of level 3 requirements on precision[1]. We require that sensors and actuators

[1] "Precision" and "accuracy" are not synonyms. For a definition and discussion of these terms, see section 3.2.3.

be accurately calibrated to a specified precision. We require that the GCS processing maintain a specified precision.

### 3.2.2 GCS Software Requirements

The refinement of the lander's requirements in the previous section shows the kind of engineering design decisions we expect would precede a software requirements document. Following our stated approach, we do not want to include in the software requirements for GCS any requirement for which the configuration of the lander or the state of the environment must be described. This leaves the following list of requirements.

- a real-time constraint on the length of each frame;
- the control laws to be implemented (these map the GCS model, i.e., the estimate of the lander's configuration, to the impulse needed to control the lander);
- an accurate calibration of each sensor (by "calibration" in this case we mean an algorithm for converting between raw sensor data and a measurement of a physical quantity);
- an accurate calibration of each actuator (by "calibration" in this case we mean an algorithm for converting between the impulse expected from the actuator and digital values sent to that actuator);
- requirements on the precision of computations;
- a requirement that the model be accurate to a specified precision.

These requirements, though derived for GCS, would apply to most control software. All but the last of these requirements can be implemented by the programmer independently of concerns about the external world. The last requirement, that the estimate of the lander's configuration be accurate, may involve facts about the external world, and for GCS it necessarily does. For this requirement, we have yet to see how software requirements can be made not to depend on a description of the controlled system.

Figure 3.2 shows the processing taking place in each GCS frame. The solid arrows represent GCS computation. The dotted arrows represent interaction with the external world. We have discussed each step in this figure except for "data fusion". In the next section we will discuss data fusion and its relationship to the requirement on the model accuracy.

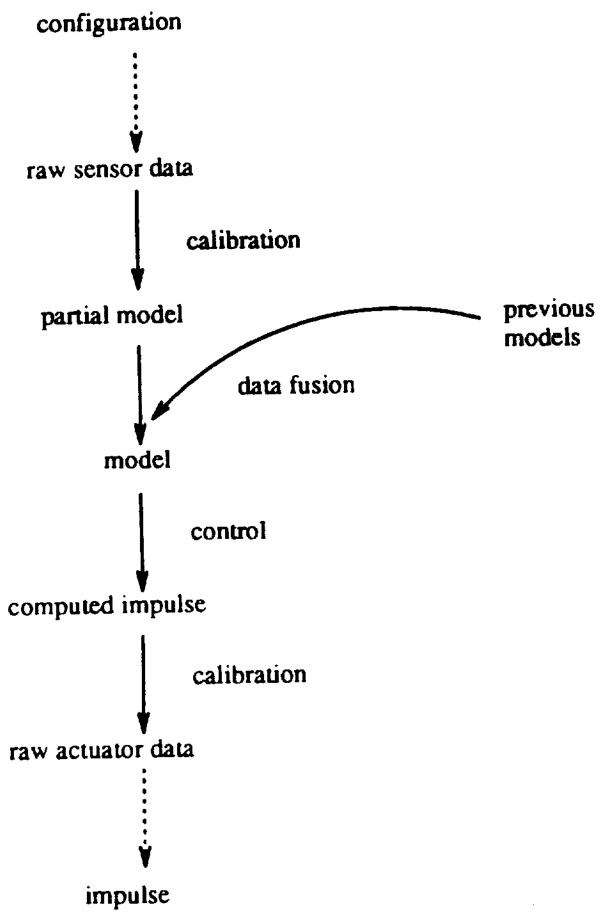
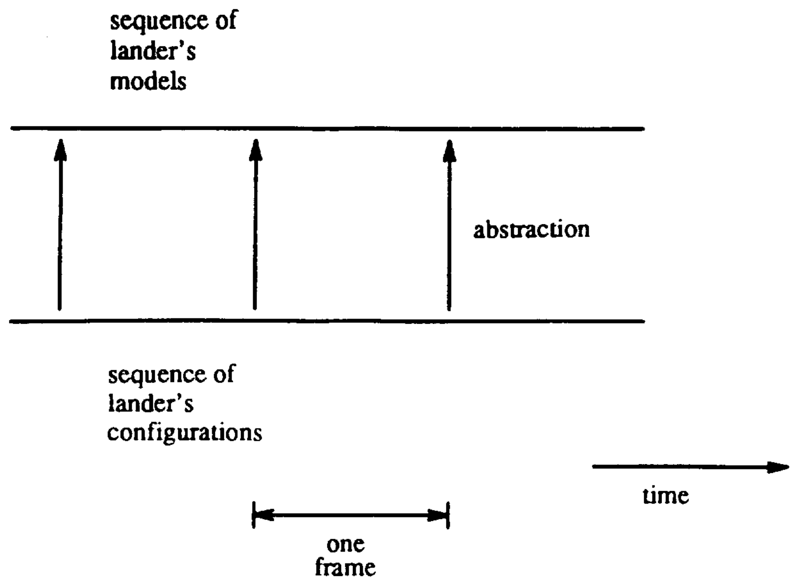


Figure 3.2: Control within one frame


sequence of lander's models
abstraction
sequence of lander's configurations
time
one frame

### 3.2.3 Data Fusion

The model computed by GCS is an estimate of the external configuration, and it is an abstraction: the model reflects those aspects of the lander's dynamics that are needed for control, while other aspects are ignored. The model is maintained throughout the lander's descent, during a sequence of frames. The maintenance of the model is diagrammed in Figure 3.2.3.

To construct the model in each frame, a control system may need to "fuse" data from different sensors. It may also need to extrapolate from a sequence of models constructed previously to get a current model, and then "fuse" the extrapolation with sensor data from the current frame.

Both of these kinds of data fusion occur in GCS. The following cases demonstrate this fact.

- Both the altitude and touch-down landing radars may fail to provide raw sensor data in a particular frame.
  - If altitude data is missing, GCS extrapolates from previous altitude measurements, making assumptions about the lander's acceleration between frames. If the extrapolation is not precise enough because previous altitude measurements were also missing, GCS integrates the velocity to yield an estimate of altitude.
  - If one touch-down landing radar value is missing out of four, GCS can still use the remaining three values to determine the three components of the lander's velocity; the four radar values are redundant. However, if more than

one value is missing, GCS integrates the acceleration to yield an estimate of velocity.

- GCS judges some acceleration measurements to be unreliable because they are too far from an extrapolation of recent measurements. In that case, GCS uses the extrapolation instead of the measurement.

- GCS fuses the temperature measurements from two different sensors to yield a single value.

In general, the purpose of data fusion in a control system is to construct a better model. But what do we mean by better?

**Accuracy and Precision**

In scientific use, the terms *accuracy* and *precision* are not synonyms. A textbook reference [1] defines them:

> The *accuracy* of an experiment is a measure of how close the result of the experiment comes to the true value. Therefore, it is a measure of the correctness of the result. The *precision* of an experiment is a measure of how exactly the result is determined, without reference to what that result means. It is also a measure of how reproducible the result is.

The concepts behind these terms are independent: a measurement can be precise but not accurate, or accurate but not precise.

We can now answer the question from the previous section. Given two models that are accurate, the better model is the one that is more precise.

### 3.2.4 Separating Concerns

GCS, like many other software control systems, maintains a model of the external world. This model is required to be both accurate and precise. Our distinction between accuracy and precision, though, allows us to separate the concerns of control software and system engineers.

Maintaining the model's precision is clearly the software engineer's concern. Given assumptions about the precision of sensors, and knowing the precision of previous models, the software engineer should be able to determine whether a given data fusion algorithm maintains the necessary precision for the model. The precision is affected both by the fusing of data with different precisions and by the finite precision of the computer.

Maintaining accuracy, on the other hand, may involve knowledge of the external world. Therefore, deciding whether a given data fusion algorithm is accurate becomes the system, not the software, engineer's problem. For example, underlying some of the data fusion in GCS is the judgement that the change in the lander's acceleration between frames is negligible. The control software engineer is not usually able to make this kind of judgement because it depends on the physics of the system.

Two approaches are possible for developing data fusion algorithms for control software, and at the same time specifying the software requirements on data fusion.

- The system engineers can simply state the data fusion algorithms as part of the software requirements. This approach has the advantage of decoupling the system and software engineering tasks. This is the approach taken to date in the GCS requirements specification.
- The system and software engineers can cooperate to specify a class of accurate data fusion algorithms, then specify a lower bound on the precision needed from these algorithms. This approach has the advantages of yielding a more abstract specification of requirements and therefore allowing the software engineer more freedom in implementing the software. It has the disadvantage that the specifications may be more difficult to express formally.

The second approach could have yielded more abstract specifications for several GCS modules. One example of this increased abstraction arises in processing touch-down landing radar (TDLR) inputs to yield a velocity measurement.

The data fusion algorithm GCS uses to determine velocity from TDLR clearly does not maximize precision. To see this, note that if only one touch-down landing radar yields a value in a particular frame, that value is ignored and integration is used instead to estimate the lander's velocity. But integration yields a value almost certainly less precise than the direct radar measurements. So combining the single radar value with the integration would give greater precision.

Maximizing precision, though, is not the goal of GCS data fusion. The goal is to satisfy a constraint on precision, along with constraints on the use of time and space by GCS. Rather than requiring a particular data fusion algorithm for TDLR, one might have stated the more abstract constraint on precision directly. Then not only would the true GCS requirements be clearer, but the GCS programmer might have chosen to implement a different algorithm, trading off precision for throughput, for example.

Our conclusion is that the GCS software requirements could have been expressed more clearly and abstractly, while still avoiding any description of the lander or its environment. However, the information about system engineering decisions needed to state bounds on precision was not available to us. Therefore we were unable to write a more abstract specification. In the next section we use the analysis of this chapter to critique the specification of requirements that was available to us.

## 3.3 Critique of Informal GCS Requirements Specification

Our formal specification of GCS requirements in Appendix A is based on an earlier, informal software requirements document [7]. We will refer to the informal software requirements document as "SR".

In the process of writing the formal specification we noted some problems with SR. Understanding most of these problems requires a detailed knowledge of GCS. These detailed problems are collected into Appendix B.

In this section we ask a more general question: how does the list of requirements in SR compare to the list in Section 3.2.2 that follows from our analysis? By comparing and noting differences in these lists, we argue that SR has the following deficiencies as a requirements specification.

- The requirements on GCS functionality are overspecified. SR specifies functionality for control, for calibrations, and for data fusion, but it goes far beyond what is needed by specifying intermediate and temporary variables and by telling in detail how the functionality is to be implemented. This extra detail makes SR essentially a software design document instead of a requirements specification.
- The requirements on GCS timing are overspecified. SR specifies the top-level timing requirement that each frame must complete within a given time. However, SR also divides the GCS functionality into modules, puts real-time requirements on each, and specifies a detailed schedule for executing these modules. The reasons underlying these extra requirements are not clear from SR. This extra detail reinforces the impression that SR is essentially a detailed design for GCS.
- The requirements on GCS precision are given too little attention. SR states *upper bounds* on precision in the following way: every variable in the design is listed in a "data dictionary", and a number of bits is specified for each variable. However, the analysis above shows that it is *lower bounds* on precision that are most important in requirements specification.

Our conclusion is that SR does not meet the goals of completeness and clarity advocated at the beginning of this chapter.

In our formal specification we have attempted to remove as much of SR's overspecification as possible. In general outline, though, our formal specification follows SR. Because SR does not provide much information about the system design decisions, we could not attempt to write a more abstract specification using the approach described in Section 3.2.4.

# Chapter 4

# Formal Methods in DO-178B

A great deal of attention is currently being focussed on the process of software engineering, and proposals for improving it. Some of that attention concerns the use of formal methods. In this chapter we consider the addition of formal methods to software engineering practice that follows the DO-178 standard for aircraft systems.

## 4.1 The Draft DO-178B Standard

The aviation community has produced a standard, DO-178B, which guides the process of engineering software for airborne systems.

The purpose of DO-178B is, in its own words

> ...to identify objectives and describe acceptable techniques and methods for the development and management of software for airborne digital systems and equipment. The application of these techniques and methods are designed to produce software that performs its intended function with a level of safety that is in compliance with airworthiness requirments and to provide evidence of compliance with those requirements.

DO-178B does not prescribe a software development method. Rather, it is a guideline for deciding on and using acceptable methods. The acceptability of a method for a particular software component depends on the criticality of that component, i.e., how hazardous would failure of the component be. The more critical the component, the greater the evidence needed that the component satisfies its requirements.

Planning the software development is a beginning step in any method acceptable under DO-178B. The planning should describe various *processes* comprising the method.

The processes are themselves composed of *activities*. There are two kinds of processes:

1. **Direct processes** are those that directly support software development. They include requirements analysis, design, coding, and integration.

2. **Integral processes** are those that support the direct processes and are ongoing throughout the software life-cycle. Examples of integral processes are verification and configuration management.

## 4.2 Including Formal Methods

Formal methods may be appropriate in a DO-178B software development plan. Formal methods encourage a disciplined approach to software development and provide evidence of higher software quality. Both these attributes of formal methods are important especially for software components that are critical to aircraft safety.

Formal specification, as we have done for GCS requirements, could be an additional activity in both the requirements analysis and design processes. Based on our GCS experience, we think the following constraints should be satisfied if formal specification is used in either process.

- The activities of informal and formal requirements specification (or design specification) should be combined. The products of these activities should be redundant, differing only in the method of expression, and so combining them should not create problems.

  Some software projects using formal methods have separated the informal activities from the formal because the latter often take longer and apparently need to be done by specialists. Separating the activities in this way, however, easily leads to inconsistencies between the formal and informal specifications, and eventually to one or the other specification becoming superfluous.

- The activity of requirements analysis for control software should be concurrent with system requirements analysis. This concurrency allows the software analysts to discuss their analysis with system engineers in order to learn the rationale and assumptions behind the requirements imposed on them.

  The alternative to this concurrent approach works much like our experience with GCS: requirements are first placed on the software, and considered finished, without a sufficient explanation of assumptions underlying them. In this situation a software specifier finds it difficult to express clear, abstract requirements of the kind discussed in Chapter 3.

# Chapter 5

# Summary

## 5.1 Conclusions

Writing a formal specif[-1zication of GCS requirements was the central activity of this task. However. the resulting specification is less important than the lessons we have learned in writing it. We summarize those lessons as follows.

- **Formal requirements specification can be routine.**

  In their foreword to the informal software requirements document for GCS [7], Dunham and Withers explain the rationale for their choice of methods:

  > [GCS] is specified using an extension to the popular method of structured analysis. This specification method was selected instead of a formal one for the sole purpose of not making the specification development activity a research effort in itself.

  This view of formal specification is mistaken. Writing the formal specification of GCS in Appendix A was straightforward. No research needed to be done. The specification was written in the Larch language, but many other specification languages would have worked as well.

  Research *is* needed, however. in the following areas:

  - understanding how to write specifications that are both clear and complete, as defined in chapter 3;
  - finding tools that support the analysis of specifications.

  Note that these research activities apply both to formal and informal methods, and therefore should not be considered special obstacles to the use of formal specification.

- **Control software requirements pose a unique problem for specification.**

  The problem is in choosing the right level of detail for the specification. Our analysis of this problem in Chapter 3 can be summarized this way: the specification must be detailed enough to be complete, yet abstract enough to be clear. When specifying a control system, abstraction naturally leads to specifications of the system being controlled and to assumptions about its environment; these are outside the domain of software engineering. In Chapter 3, we argue that a balance must be struck between completeness and abstraction. We also argue that if more detail about the system engineering behind the GCS lander had been available, we could have written our formal specification more clearly while still avoiding descriptions of the lander and its environment.

- **Formal methods can be used to augment current software development processes.**

  Chapter 4 explains how formal specification could be incorporated into DO-178B, and lists some constraints on software development processes using formality.

## 5.2 Extensions to this Work

The natural steps to take after specifying a system's requirements are implementing and verifying that system. If GCS were implemented in Ada, a Larch/Ada interface specification could easily be written from the specification in Appendix A. The leading software tool for verifying Larch/Ada specifications is ORA's Penelope environment [5].

# Appendix A

# Formal Specification of GCS

## A.1 Introduction

This chapter contains the formal requirements specification for NASA's Guidance and Control Software (GCS).

GCS controls the descent of a spacecraft onto a planet's surface. The spacecraft is a simplification of the 1970s Viking Mars lander. The lander is simplified because GCS is being developed as a study in software methods. By simplifying the lander, GCS requirements, design, and code are all shortened, making possible a study in which GCS is implemented several times using different methods. GCS development using formal specification is one method under study.

Our specification is derived from an informal software requirements document for GCS, "Software Requirements: Guidance and Control Software Development Specification", written by staff at RTI[7]. We will refer to this earlier document as "SR".

SR uses a variant of structured analysis adapted for real-time systems. Structured analysis shows the decomposition of a system by a hierarchy of diagrams. The functionality of each element at the bottom of the hierarchy is described by some informal text.

Our specification is written in Larch. We give a brief introduction to Larch in the next section, then we prepare the reader by describing the intent and organization of the Larch specification. The Larch specification is divided into modules roughly corresponding to those in SR.

Although our specification follows SR, we have tried to avoid including unnecessary design details from SR. To do this we have sometimes made educated guesses about the intent underlying the SR requirements. Section 3.3 explains how SR is unnecessarily detailed for a requirements specification. Appendix B lists problems we encountered with SR in the process of formalizing the requirements.

### A.1.1 Larch

Larch divides a formal specification into two parts: a mathematical part and an interface part. The mathematical part describes the theory underlying the symbols used in a specification. The interface part uses the mathematical theory to specify the properties and behavior of a system. The interface part is the connection between the mathematical theory and some language (such as Ada or C) used to implement systems.

There is a single language for the mathematical part, called the *Larch Shared Language* (LSL). Because there are many implementation languages in use, however, there can be many interface languages; each such language is called a *Larch Interface Language.* For example, ORA is developing an interface language for Ada called Larch/Ada. As another example, the tools used to process this specification recognize an interface language. Larch/C, for interfacing to C programs; these tools are distributed by Digital Equipment Corporation.

This division of specifications into two parts is called the Larch *two-tiered* approach. The philosophy underlying this approach is that most of a formal specification should be expressed in LSL because

- its meaning is then independent of subtleties of programming language semantics;
- it is portable between implementation languages.

The GCS specification here is written entirely in LSL. Once an implementation language is chosen for GCS, though, a small part of the LSL will need to be re-written as an interface specification for the implementation.

The basic form of an LSL specification is sketched in the next few sections. This introduction should be enough to understand the GCS specification because the design of the Larch language is quite simple. The Larch terminology may be unfamiliar because it uses some terms for familiar concepts in unfamiliar ways in order to avoid confusion with terms from programming languages. Larch is not a programming language, and so the tendency to think computationally about a specification should be resisted.

More detail on LSL is available in the published reports on Larch [6].

## Sorts and Operators

The mathematical objects described by an LSL specification form collections called *sorts*. Every object is of some sort. There are no special declarations of sorts in Larch; each sort in a specification is given a name and properties by the introduction of operators that use the sort.

An *operator* is a mathematical function. Each operator has a signature that tells the sort of the operator and the sorts of its arguments. Operators are declared after the keyword **introduces**. For example,

$$\begin{array}{l}\textbf{introduces}\\ exponential : Real, Nat \rightarrow Real\end{array}$$

declares an operator *exponential* that maps a pair of objects, of sorts *Real* and *Nat*, respectively, into an object of sort *Real*.

The properties of sorts and operators are specified after the keyword **asserts**. These properties are expressed in several ways.

- Equations involving constants, e.g., $1 + 1 == 2$, follow the **equations** keyword.
- Equations involving logical variables follow a universal quantifier. For example:

  $$\begin{array}{l}\forall r : Real, n : Nat\\ \quad exponential(r, 0) == 1\\ \quad exponential(r, n + 1) == r * exponential(r, n)\end{array}$$

  express properties of the exponential operator introduced earlier. These properties are equations that hold for every ($\forall$) pair of objects, $r$ and $n$, of the stated sorts.
- A clause of the form *Nat* **generated by** *zero, successor* means that every object of the sort *Nat* can be formed by the application, possibly repeated, of the operators *zero* and *successor*. If *Nat* is meant to specify the natural numbers, we would write this clause to state that every natural number is in the sequence

  $$\begin{array}{l}0 == zero()\\ 1 == successor(zero())\\ 2 == successor(successor(zero()))\end{array}$$

  and so on. This clause would be useful in proving properties that are true for every natural number.
- A clause of the form *Complex* **partitioned by** *real, imaginary* means that two objects of sort *Complex* are distinguishable only if either the *real* operator (not

the same as the *Real* sort used previously), or the *imaginary* operator, or both, yield different values when applied to the *Complex* object. If *Complex* is meant to specify the complex numbers, this clause means that two complex numbers are equal when their real and imaginary parts are equal.

## Traits

An LSL specification can be made modular. Each module of the specification is a separate mathematical theory called a *trait*. Each trait introduces some operators and asserts some properties about them, as described previously.

Traits can be combined. If trait $A$ contains the clause **includes** $B$, then the meaning of $A$ is as if the **includes** clause had been replaced by the text of $B$.

When trait $B$ is included, any sort or operator appearing in it can be renamed. Some sorts and operators are shown as parameters in the trait declaration; these must be renamed (possibly to themselves). The clause **includes** $B(foo, bar, x$ **for** $y)$ is an example in which $foo$ and $bar$ are the new names for the trait parameters, and symbol $x$ is the new name for $y$.

If trait $A$ contains the clause **assumes** $B$, then the meaning of $A$ is as for an **includes** clause, but the specifier also insists that every property specified in $B$ be provable in any trait that includes $A$. This is useful when the parameters of trait $A$ need to have particular properties for $A$ to make sense.

## Interface Langauges and Abstraction

Usually a Larch interface language allows one to specify pre- and post-conditions of a computation. For example, one might like to specify that, if a programming language procedure $P(x, y)$ is called, and $pre(x, y)$ is true at the time of the call, and $P$ terminates, then $post(x, y)$ is true after the call. In this case, *pre* and *post* are LSL operators whose meaning is given in some LSL trait. Using LSL to specify $P$ in this way allows one to reason purely mathematically about the effect of calling $P$, without worrying about the sequencing of events that must happen when $P$ is run on a computer.

Note that when $x$ and $y$ are arguments to procedure $P$, they name programming language objects, but when they are arguments to operators *pre* and *post* they must name objects of Larch sorts. This dual use is important for specification. In order for a Larch interface specification to make sense, it must describe the connection between the types of programming langauge objects implemented on a computer and the mathematical sorts these types are supposed to model. Once this correspondence is made, the arguments to procedure $P$ can be interpreted as mathematical objects of the corresponding sort.

When a Larch sort is used to specify an abstract data type, an *abstraction function* is needed to map concrete program objects into the sort. Abstraction functions appear in the GCS specification in only one place: to relate the concrete design of I/O registers to abstract sorts used in the rest of the specification. Because we have chosen to write the entire specification in LSL, even the I/O registers are specified using sorts. However, if a true interface specification for GCS were written, these sorts would be replaced by types in the programing language.

### A.1.2 Organization

Chapter 3 analyzes the difficulties in specifying control software. One result of that analysis is that our GCS specification is limited to software requirements that can be stated without reference to spacecraft dynamics. For example, we will never specify the accuracy of sensor processing, i.e., how close a computed value is to the actual, real world value. We might specify the precision of a processed value, although this too sometimes depends on assumptions about the real world. Usually we simply specify abstract functionality. This limitation on the scope of software requirements is needed to separate the concerns of the system engineer from those of the software engineer. The effect can be to make the software requirements specification relatively close to the design, and not very abstract. Ideally, the software requirements would be used as partial justification that more abstract requirements on the entire system are met.

GCS processing is divided into a sequence of *frames*. Each frame processes new sensor data, updates an internal state, sends a snapshot of the internal state back to an orbiting platform, and on the basis of the internal state controls actuators that affect the lander's flight. Each frame must be completed within a fixed duration.

Our specification is a requirement on a single arbitrary frame. The top-level trait, Frame. describes the change of state and control that can happen during a frame, given particular sensor inputs. The Communication trait specifies the values to be sent out as a function of the frame's state.

Preceding the Frame trait is the GCS trait, which specifies a collection of basic sorts used throughout the specification.

Following the Frame and Communication traits, there are eight sections describing various kinds of processing to be done within one frame. Most of these sections contain more than one trait. Our division into these sections corresponds closely to the modules described in SR. Minor modules from SR have been folded into other modules or into the top-level specification. We have consistently used the following acronyms to distinguish the eight (following SR):

- AE: Axial Engines
- RE: Roll Engines
- A: Acceleration
- AR: Altimeter Radar
- G: Gyroscope
- T: Temperature
- TDLR: Touch-Down Landing Radar
- GP: Guidance Processing

All but the last of these sections specify processing related to a particular part of the lander's hardware. See chapter 2 for a more detailed explanation of the lander and GCS.

A key idea underlying this specification is that GCS constructs an estimate of the lander's dynamical state, i.e, its position, velocity, etc. This estimate is based on sensor data. One high-level requirement is that this estimate be as accurate as possible. We do not state this requirement because it involves the system outside the software. When there is more than one way to compute a dynamical state variable, however, the requirement is that the more precise method is chosen. This choice must be made for altitude, velocity, and for temperature estimates. The function of "Guidance Processing" is to make some of these choices.

Following the eight main traits is an interface specification trait showing the relation between I/O registers and abstract sorts used to specify I/O. Next, are several traits that are GCS-specific but ancillary.

The specification concludes with a dozen or so traits that are used in the GCS specification but describe theories that would also be useful in other contexts.

The specification refers to some traits that are not shown here. These traits are Set, Bag, Cardinal, Field, Ring, TotalOrder, AbelianSemigroup, and Distributive. Each of these traits axiomatizes basic mathematical concepts. Each is included with the version of DEC's Larch/C tools we used to process the specification.

Wherever possible we have used names for state variables, parameters, and operations that match those in SR. However, the correspondence is not exact because we have aimed to represent data abstractly, and to eliminate inessential detail.

Our specification is written purely in LSL. Instead of our Frame and Communication traits, a Larch specifier would normally write an Interface Language specification that constrains procedures written in some implementation language. We did not do this because

- we did not want to choose an implementation language yet;
- our tools for processing LSL specifications are better than our tools for processing any Larch Interface Language;
- these traits were simple enough that converting them to an interface specification later would be easy.

The specification has been processed by DEC's Larch/C tools. These tools are not yet production quality. For example, the LaTeXoutput of the tools doesn't control indentation very well. The reader will see that this detracts from the clarity of the specification in some places.

### A.1.3 Terminology and Conventions

In the specification, all vector operations are described using the lander's coordinate system. This is a right-handed system in which the positive x-axis is toward the bottom of the lander. *Roll* is rotation around the x-axis; *pitch* is rotation around the y-axis; *yaw* is rotation around the z-axis. In each case, positive rotation is right-handed.

## A.2 Top Level Specification

### A.2.1 Basic Sorts

% Define sorts used commonly in GCS traits.
% These sorts include vectors, tensors, and histories.
% We do not explicitly specify real time as part of the state,
% but the real-time requirement is simply that each frame complete
% within delta_t.

*GCS* : **trait**

**includes** *Real*
**includes** *Natural*
*sense* **enumeration of** *clockWise*, *counterClockWise*

*coordinate* **enumeration of** $x, y, z$
**includes** *Vector*(*RealVector*, *Real*)
**includes** *Tensor*(*RealTensor*, *Real*, *RealVector* **for** *vector*)
**includes** *Rotation*

**includes** *Array*(*NatTriple*, *coordinate*, *Nat*)
**includes** *Map*(*BoolTriple*, *coordinate*, *Bool*)
**includes** *Map*(*senseTriple*, *coordinate*, *sense*)

**includes** *History*(*RealHistory*, *Real*)
**includes** *History*(*RealVectorHistory*, *RealVector*)
**includes** *History*(*BoolHistory*, *Bool*)
**includes** *History*(*BoolTripleHistory*, *BoolTriple*)

% The lander's descent passes through several phases.
% The phases are differentiated by the temperature of the engines,
% whether the lander has crossed (and is currently following)
% the velocity-altitude contour, and whether the lander is close
% enough to the ground to shut off engines and drop the rest of the way.

%
*engineTemp* **enumeration of** *cold*, *warming*, *hot*
*phase* **tuple of**
  *engine* : *engineTemp*,
  *contour* : *Bool*,
  *landing* : *Bool*

**introduces**
  *delta_t* :→ *Real* % duration of each frame

## A.2.2 State Machine Specification

% GCS processing is a sequence of frames.
% Each frame processes sensor data, updates an internal state, and
% controls actuators in real time.
% GCS processes the sensor data into an estimate of the lander's physical
% situation: its position, velocity, etc.
% The internal state records this estimate, along with other information
% needed in later frames.
% Based on the internal state, GCS sets the actuators to control flight.
%
% The processing in an arbitrary frame is modeled here using LSL;
% this model can be trivially translated into a Larch Interface Language
% specification for a Frame procedure coded in some programming language.
%
% The representation of sensor and actuator data, and of internal state,
% is abstract, which means that not all details of the physical I/O registers,
% and not all details of the actual internal state are shown.
% The connection to the concrete representation in input registers
% is given in the Interface trait by an abstraction function.
% The concrete internal state, and an abstraction function connecting it with
% the abstract state, must be supplied by the GCS programmer.

*Frame* : **trait**

**includes** *GCS* % basic GCS sorts

```
includes ARSP, ASP, GSP, TDLRSP, TSP, GP                % sensor processing
includes AECLP, RECLP                                    % actuator control

   % The components of this tuple represent sensor readings for each frame.
   %
sensor tuple of

   AR_counter : Nat,                              % time for altitude radar echo
   AR_counter_OK : Bool,                                          % echo heard?
   A_counter : NatTriple,                  % accelerometer reading for each axis
   G_counter : NatTriple,                      % gyroscope reading for each axis
   G_counter_sense : senseTriple,            % clock-wise or counter-clock-wise?
   TDLR_counter : NatQuad,                % frequency shift for 4 doppler radars
   TDLR_counter_OK : BoolQuad,                   % shift available in this frame?
   SS_temp : Nat,                                % solid-state temp sensor reading
   Thermo_temp : Nat,                          % thermocouple pair temp reading
   touch_down : Bool                                          % surface touched?

   % The components of this tuple represent commands
   % to actuators for each frame.
   %
actuator tuple of

   AE_cmd : NatTriad,                          % valve settings for 3 axial engines
   RE_cmd_intensity : REintensity,                % intensity of roll engine pulse
   RE_cmd_sense : sense,                                      % roll which way?
   release_chute : Bool                        % end parachute phase of descent

   % The abstract state of GCS.
   %
state tuple of

   % Represent the lander's external dynamics.
   % We need histories for integration and for extrapolation of the dynamics.
   altitude : RealHistory,
   velocity : RealVectorHistory,
   acceleration : RealVectorHistory,
   spin : RealVectorHistory,
   temperature : Real,
    % Record whether altitude data is missing, and
    % whether acceleration data is bad.
    AR_OK : BoolHistory,
    A_OK : BoolTripleHistory,

    % Record the progress of the lander's descent (the phase),
    % and some convenient internal variables.
    phase : phase,
    frames_engines_warming : Nat,
    frame_counter : Nat,
    integrated_thrust : Real

introduces
    % Declare two operators that specify the requirements on an arbitrary frame:
    % frameUpdate yields a new state from the old state plus the current input.
    % frameControl yields the output from the old and new states.
    %
    frameUpdate : sensor, state → state
    frameControl : state → actuator

    % The state must be initialized when GCS is turned on.
    initialize :→ state

    % Because of finite precision, no computer can exactly compute the results
    % (e.g., of sort Real) specified by frameUpdate and frameControl.
    % Therefore, the next two functions specify whether the computed results
    % are acceptably close to the specified results.
    % SR does not give any details about these functions.
    updatePrecision : state, state → Bool
    controlPrecision : actuator, actuator → Bool

    % These constants are used to determine transitions between phases of descent.
    engines_on_altitude :→ Real               % when should warm-up start?
    full_up_time :→ Real                  % how long before engines are hot?
    drop_height :→ Real                             % when have we landed?

asserts
    % frameUpdate, frameControl, and initialize are specified here.
    % All other operators used in their specification are defined by traits
    % included in this one.
% In particular, the operator update, defined in the History trait,
% extends an old history into a new one by adding the latest value.
%
equations
    initialize.phase == [cold, false, false]
    initialize.frames_engines_warming == 0
    initialize.frame_counter == 0
    initialize.integrated_thrust == 0
    % SR does not specify how the dynamical histories
    % are to be initialized. Presumably the following will be needed
    % to prevent spurious extrapolations right after start-up:
    now(initialize.AR_OK) == false
    now(initialize.A_OK)[x] == false
    now(initialize.A_OK)[y] == false
    now(initialize.A_OK)[z] == false


∀ in : sensor, old, new : state
    frameUpdate(in, old) = new ==


        new.altitude = update(old.altitude,
        altitude(in.AR_counter,
        in.AR_counter_OK,
        old.altitude,
        old.AR_OK,
        new.velocity,
        new.spin))
        ∧
        new.velocity = update(old.velocity,
        velocity(in.TDLR_counter,
        in.TDLR_counter_OK,
        old.velocity,
        new.acceleration,
        new.spin))
        ∧
        new.acceleration = update(old.acceleration,
        acceleration(in.A_counter,
        new.temperature,
        old.acceleration,
        old.A_OK))
        ∧
        new.spin = update(old.spin,
spin(in.G_counter,
in.G_counter_sense,
new.temperature))
∧
new.temperature =
temperature(in.SS_temp,
in.Thermo_temp)
∧
new.AR_OK = update(old.AR_OK,
in.AR_counter_OK)
∧
new.A_OK = update(old.A_OK,
accelerationOK(in.A_counter,
new.temperature,
old.acceleration,
old.A_OK))
∧
new.phase.landing =
(old.phase.landing∨
now(new.altitude) < drop_height∨
in.touch_down)
∧
new.phase.contour =
(old.phase.contour∨
now(speedError(new.altitude, new.velocity)) > 0)
∧
new.phase.engine =
( if old.phase.engine = hot∨
(old.phase.engine = warming∧
((natToReal(old.frames_engines_warming) * delta_t)
≥ full_up_time))
then hot
else if old.phase.engine = warming∨
(now(new.altitude) < engines_on_altitude)
then warming
else cold)
∧
new.frames_engines_warming =
( if new.phase.engine = warming
then old.frames_engines_warming + 1
else 0)
∧
```

$new.frame\_counter =$
$old.frame\_counter + 1$
$\wedge$
$new.integrated\_thrust =$
$eThrust(eVel(currentSpeed($
$new.altitude,$
$new.velocity,$
$new.acceleration)),$
$old.integrated\_thrust)$

$\forall\ new : state, out : actuator$
$frameControl(new) = out ==$

$out.AE\_cmd =$
$AEcommand(new.altitude,$
$new.velocity,$
$new.acceleration,$
$new.spin,$
$new.integrated\_thrust,$
$new.phase)$
$\wedge$
$out.RE\_cmd\_intensity =$
$REintensity(new.spin, new.phase)$
$\wedge$
$out.RE\_cmd\_sense =$
$REsense(new.spin)$
$\wedge$
$out.release\_chute =$
$(out.release\_chute \vee new.phase.engine = hot)$

## A.2.3 Communications

% At least once per frame, GCS radios a snapshot of its state to an orbiter.
% The snapshot is contained in a message packet, which also contains
% a synchronization pattern, the frame counter, and a checksum.
% The snapshot contains variables describing the lander's dynamical state,
% but also includes other internal variables of the software engineer's
% choosing.

*Communication* : **trait**

**includes** *Frame*

*packet* **tuple of**
  *pattern* : *Nat*, % synchronization pattern
  *number* : *Nat*, % a function of frame count

  *altitude* : *Real*,
  *velocity* : *RealVector*,
  *acceleration* : *RealVector*,
  *spin* : *RealVector*,
  *temperature* : *Real*,

  *internal* : *setOfVariables*,

  *checksum* : *Nat* % a cyclic redundancy check

**introduces**
  % Every packet sent is a function of the current state.
  *commUpdate* : *state* $\rightarrow$ *packet*

  % The synchronization pattern is constant, depending only on
  % the communication hardware. The cyclic redundancy check (CRC)
  % depends on the entire packet except for the CRC field itself.
  *synchronizationPattern* : $\rightarrow$ *Nat*
  *CRC* : *packet* $\rightarrow$ *Nat*

**asserts**
  $\forall$ *s* : *state*

$$
\begin{aligned}
&commUpdate(s).pattern == synchronizationPattern\\
&commUpdate(s).number == mod(s.frame\_counter, 256)\\
&commUpdate(s).altitude == now(s.altitude)\\
&commUpdate(s).velocity == now(s.velocity)\\
&commUpdate(s).acceleration == now(s.acceleration)\\
&commUpdate(s).spin == now(s.spin)\\
&commUpdate(s).temperature == s.temperature\\
&commUpdate(s).checksum == CRC(commUpdate(s))
\end{aligned}
$$

## A.3 Sensor Processing

The high-level goal for processing sensor inputs is to model the lander's dynamical state, including its altitude, velocity, acceleration, angular velocity, and temperature. Values in the model are estimates of these dynamical quantities. The model values must be both accurate and precise.

Accuracy depends on making the correct conversion from raw sensor data to model values. This conversion uses facts about each physical sensor, and includes the proper calibration values and adjustments for the environment (temperature, misalignment, etc.). Deriving the details of the conversion is clearly not the software engineer's problem. The software requirement, then, is simply a function describing the conversion.

Precision, on the other hand, depends on these factors:

1. the precision of the raw sensor data;
2. the reliability of the sensors;
3. propagation of errors during calcuation;
4. the precision of machine arithmetic;
5. assumptions about the dynamical state that permit current values to be derived by extrapolation or integration from previous values.

The software engineer has some, but not complete, control over precision. The first, second, and last of these factors are outside his control. The fourth is not. The third factor, propagation of errors, is partly determined by the software engineer but is outside his control to the extent that calcuations are determined by the accuracy requirement.

What form should the requirement on precision take? Here are the extremes:

- The requirements could specify the design in great detail, including all the calculations to be performed and the precision of variables in which intermediate results are to be stored.
- The requirements could specify the precision of the result, make the assumptions about sensors and the environment explicit, and let the software engineer choose the data structures and algorithms that satsify both the precision and accuracy requirements.

The document on which this specification is based, SR, takes the first approach. In the sensor processing traits we will indicate ways in which more information about the assumptions underlying SR might allow the software engineer more freedom to design the GCS software.

### A.3.1 Altitude Radar

```
% Altitude Radar Sensor Processing (ARSP)
%
% The primary means for the lander to determine its altitude is by timing
% the echoes of radar pulses bounced off the planet's surface.
% Some echoes might not be detected, in which case an altitude estimate
% can be found by extrapolating from previous measurements.

ARSP : trait

  includes ARSPauxil

  introduces
    % The altitude depends on a count of the time between pulse sent and
    % echo received (Nat), whether the echo was received at all (Bool),
    % and the history of previous altitude measurements (RealHistory).
    altitude : Nat, Bool, RealHistory → Real

  asserts
    ∀ count : Nat, echo : Bool, hist : RealHistory
      altitude(count, echo, hist) ==
        if echo then countToDistance(count)
        else value(determinePolynomial(altitudeHistory(hist)), 0)
```

```
% If an echo is not detected in the current frame, how can we know
% that an extrapolation from previous data is precise enough?
% SR seems to assume that
% 1) the probability of missing an echo is small, so that there is a
% decent chance there have been measured values in each of the previous
% four frames;
% 2) the derivatives of altitude are small, so an extrapolation based
% on a cubic polynomial fit will be both accurate and precise.
%
% Why just four frames?
% Fewer frames might decrease the chance of missing values;
% more frames might increase precision.
%
% Why fit a cubic?
% If the derivative of acceleration is negligible, fitting a quadratic
% might be more accurate; if the acceleration is negligible over four
% frames. a linear fit might be best.
```

*ARSPauxil* : **trait**

  **includes** *GCS*
  **includes** *PolynomialFit(Nat, Real, natToReal,*
    *measurements* **for** *pairSet)*

  **introduces**
    % Whether the current altitude value is reliable enough to use depends on
    % whether the echo was received (Bool) and whether enough echoes
    % have been received in recent frames to extrapolate
    % to the current altitude (BoolHistory).
    *altitudeOK* : *Bool, BoolHistory* → *Bool*

    % The raw sensor value is proportional to distance.
    *countToDistance* : *Nat* → *Real*

```
    AR_frequency, speedOfLight :→ Real

    % Extrapolation is based on a set of recent altitude measurements.
    altitudeHistory : RealHistory → measurements

asserts
    ∀ echo : Bool, hist : BoolHistory
        altitudeOK(echo, hist) ==
            echo∨
            (hist[0] ∧ hist[1] ∧ hist[2] ∧ hist[3])

    ∀ count : Nat
        countToDistance(count) ==
            (natToReal(count) * speedOfLight)/AR_frequency

    ∀ hist : RealHistory
        altitudeHistory(hist) ==
            {[1, hist[0]], [2, hist[1]], [3, hist[2]], [4, hist[3]]}
```

### A.3.2 Accelerometers

```
% Accelerometer Sensor Processing (ASP)
%
% The lander has three accelerometers, one for each coordinate axis.
% ASP converts the readings from these sensors into an acceleration vector
% by scaling, adjusting for temperature, and correcting for misalignment.
% The readings have a small but significant probability of being very
% inaccurate, so ASP filters the data by comparing each measurement with
% previous ones and replacing any flaky value with an average.

ASP : trait

   includes ASPauxil

   introduces
      % Acceleration depends on the raw sensor values (NatTriple),
      % the temperature (Real), the acceleration history (RealVectorHistory),
      % and the history of whether an acceleration component is based on
      % a measurement or is an average (BoolTripleHistory).
      acceleration : NatTriple, Real,
      RealVectorHistory, BoolTripleHistory → RealVector

   asserts
      ∀ count : NatTriple, temp : Real, i : coordinate,
         ah : RealVectorHistory, bh : BoolTripleHistory

         % If recent measurements are reliable, force the current measurement
         % to be the average of recent measurements.
         acceleration(count, temp, ah, bh)[i] ==
            if accelerationOK(count, temp, ah, bh)[i]
            then adjustedAccel(count, temp)[i]
            else mean(recent(ah, i))
```

% How can we guarantee the current acceleration value has the required
% precision? By comparing with recent measurements,
% by assuming that changes to acceleration are negligible between frames,
% and by assuming that the probability of a flaky measurement in any given
% frame is small.
%
% SR prescribes replacing a flaky measurement with an average
% of the most recent three acceleration values,
% but only if each of the three was itself near the mean.
% This ad hoc method leads to the funny situation that the current
% measurement is considered more reliable if its predecessors are less so.
%
% Why three values?
% Why not use several of the most recent uncorrected values?

*ASPauxil* : **trait**

**includes** *GCS*
**includes** *Statistics*

**introduces**
% Whether the current acceleration components are based on measurement
% or on an average, depends on the same factors as does the acceleration.
*accelerationOK* : *NatTriple*, *Real*,
*RealVectorHistory*, *BoolTripleHistory* $\rightarrow$ *BoolTriple*

% Convert raw values to an acceleration vector.
% The conversion gain, A_gain, depends on temperature.
*measuredAccel* : *NatTriple*, *Real* $\rightarrow$ *RealVector*
*A_gain* : *Real* $\rightarrow$ *RealVector*
$A\_gain_0$, *A_bias* :$\rightarrow$ *RealVector*
$g_1, g_2$ :$\rightarrow$ *Real*

% Adjust for misalignment of the accelerometers.
% The adjustment is expressed by a constant tensor.
*adjustedAccel* : *NatTriple*, *Real* $\rightarrow$ *RealVector*
*alpha_matrix* :$\rightarrow$ *RealTensor*

% Are recent measurements good enough for comparison with current?
*recentOK* : *BoolTripleHistory* $\rightarrow$ *BoolTriple*

% What is the range of believable acceleration values?
% The range is a constant multiple of the standard deviation
% of recent acceleration values.
*minAccel*, *maxAccel* : *RealVectorHistory* → *RealVector*
*A_scale* :→ *Real*

% Put the recent acceleration values into a bag.
*recent* : *RealVectorHistory*, *coordinate* → *RealBag*

**asserts**
$\forall$ *count* : *NatTriple*, *temp* : *Real*, *i* : *coordinate*,
*ah* : *RealVectorHistory*, *bh* : *BoolTripleHistory*

% The current acceleration value is considered reliable
% if it couldn't be, or didn't need to be, forced near the average.
$accelerationOK(count, temp, ah, bh)[i] ==$
$(\neg(recentOK(bh)[i]))\vee$
$between(minAccel(ah)[i],$
$adjustedAccel(count, temp)[i],$
$maxAccel(ah)[i])$

$\forall$ *count* : *NatTriple*, *temp* : *Real*, *i* : *coordinate*
$A\_gain(temp)[i] ==$
$A\_gain_0[i]+$
$((g_1 * temp)+$
$(g_2 * (temp\hat{}2)))$
$measuredAccel(count, temp)[i] ==$
$A\_bias[i] + (A\_gain(temp)[i] * natToReal(count[i]))$

$\forall$ *count* : *NatTriple*, *temp* : *Real*
$adjustedAccel(count, temp) ==$
$alpha\_matrix \otimes measuredAccel(count, temp)$

$\forall$ *h* : *BoolTripleHistory*, *i* : *coordinate*
$recentOK(h)[i] ==$
$(h[0])[i] \wedge (h[1])[i] \wedge (h[2])[i]$

$\forall$ *h* : *RealVectorHistory*, *i* : *coordinate*

```
recent(h, i) ==
    {(h[0])[i], (h[1])[i], (h[2])[i]}
minAccel(h)[i] ==
    mean(recent(h, i))−
    (A_scale * standardDeviation(recent(h, i)))
maxAccel(h)[i] ==
    mean(recent(h, i))+
    (A_scale * standardDeviation(recent(h, i)))
```

### A.3.3 Gyroscopes

```
% Gyroscope Sensor Processing (GSP)
%
% The lander has three gyroscope sensors, each permitting a measurement
% of the rate of rotation around an axis.
% There is one gyroscope sensor for each of the lander's coordinate axes.
% GSP converts the raw sensor data into a angular velocity psuedo-vector
% by scaling, and by adjusting for temperature.
%
% For brevity. we're also calling the angular velocity "spin".

GSP : trait

    includes GSPauxil

    introduces
        % The spin vector depends on three measurements of rotation speed
        % (NatTriple), three measurements of rotation direction (senseTriple),
        % and temperature (Real).
        spin : NatTriple, senseTriple, Real → RealVector

    asserts
        ∀ count : NatTriple, dir : senseTriple, temp : Real, i : coordinate
            spin(count, dir, temp)[i] ==
                if dir[i] = counterClockWise
                then rate(count, temp)[i]
        else $-rate(count, temp)[i]$

        % SR is unclear whether the sign is part of the rate calculation or not.


GSPauxil : trait

  includes GCS

  introduces
    % The rotation rate around each axis is independent of direction,
    % depending only on speed and temperature (unclear from SR).
    $rate : NatTriple, Real \rightarrow RealVector$
    $G\_offset : \rightarrow RealVector$

    % The gain for converting from counter values to speed depends on
    % temperature.
    $G\_gain : Real \rightarrow RealVector$
    $G\_gain_0 : \rightarrow RealVector$
    $g_3, g_4 : \rightarrow Real$

  asserts
    $\forall\ temp : Real, i : coordinate$
      $G\_gain(temp)[i] ==$
        $G\_gain_0[i] +$
        $((g_3 * temp) +$
        $(g_4 * (temp\hat{}2)))$

    $\forall\ count : NatTriple, temp : Real, i : coordinate$
      $rate(count, temp)[i] ==$
        $G\_offset[i] +$
        $(G\_gain(temp)[i] * natToReal(count[i]))$
```

### A.3.4 Touch Down Landing Radar

% Touch Down Landing Radar Sensor Processing (TDLRSP).
%
% The lander has four Doppler radars, each of which measures the vehicle's
% speed in a single direction: along one radar beam.
% Each speed is the projection of the vehicle's velocity vector along
% the beam direction.
% TDLRSP combines the speed measurements into an estimate of the velocity.
%
% The raw sensor data for each radar consists of a counter value and a
% boolean indicating whether the count is meaningful in this processing frame.
% So there may be anywhere from 0 to 4 speed measurements available in a frame.
% The velocity vector has three components, which will be determined
% by three measurements of speed in independent directions.
% Therefore the velocity may be either overdetermined or underdetermined,
% depending on the number of speed measurements.
% TDLRSP takes an average if four speeds were measured.
% If fewer than three speeds were measured, TDLRSP reports which velocity
% components have been determined (some may be determined even if not all are
% because the radars are symmetrically placed in the vehicle coordinate
% system, in which the velocity is to be calculated).

*TDLRSP* : **trait**

**includes** *TDLRSPauxil*

**includes** *Matrix*(*RealMatrix*, *coordinate*, *Real*, *RealVector* **for** *array*)
**includes** *LinearEquation*(*RealVector*, *RealMatrix*, *coordinate*, *Real*,
*BoolTriple* **for** *BoolArray*)

**introduces**
% The raw sensor data will be a NatQuad and a BoolQuad.
% From this data the following two operators extract
% 1) the velocity vector and
% 2) a boolean for each vector component in the vehicle coordinates,
% telling whether the component has been uniquely determined.
*velocity* : *NatQuad*, *BoolQuad* → *RealVector*
*velocityOK* : *NatQuad*, *BoolQuad* → *BoolTriple*

% An average of four determinations of velocity is defined as

```
% a component-wise weighted average.
% Each weight is 1 if its corresponding velocity component is determined,
% 0 if it is underdetermined.
% Two auxiliary operators are defined:
% a weighted product, and the sum of the weights.
average : RealVectorQuad, RealVectorQuad → RealVector
Π : RealVectorQuad, RealVectorQuad → RealVector
Σ : RealVectorQuad → RealVector

% Four constraints, one for each beam, can be used to determine
% three velocity components in four ways by ignoring each
% beam in turn.
% vel is the quad of solutions gotten in this way.
% OK is a matrix of weights: 1 if a vector component is determined,
% 0 otherwise.
vel : NatQuad → RealVectorQuad
OK : NatQuad, BoolQuad → RealVectorQuad

% Ignoring one of the four beams yields triples of vectors.
% To express linear equations having these triples as coefficients,
% we must convert vector triples to matrices.
vectorTripleToMatrix : RealVectorTriple → RealMatrix
```

**asserts**

```
  ∀ nq : NatQuad, bq : BoolQuad, i : coordinate
      velocity(nq, bq) == average(vel(nq), OK(nq, bq))
      velocityOK(nq, bq)[i] == Σ(OK(nq, bq))[i] > 0

  ∀ nq : NatQuad, b : beam
      vel(nq)[b] ==
          solve(ignore(b, beamVelocity(nq)),
          vectorTripleToMatrix(ignore(b, TDLR_angles)))

  ∀ nq : NatQuad, bq : BoolQuad, b : beam, i : coordinate
      (OK(nq, bq)[b])[i] ==
          if determine(ignore(b, bq),
          ignore(b, beamVelocity(nq)),
          vectorTripleToMatrix(ignore(b, TDLR_angles)))[i]
          then 1
          else 0
```

```
∀ value, weight : RealVectorQuad, i : coordinate
    average(value, weight)[i] ==
        Π(value, weight)[i]/Σ(weight)[i]
    Π(value, weight) ==
        ((value[1] * weight[1])+
        (value[2] * weight[2])+
        (value[3] * weight[3])+
        (value[4] * weight[4]))
    Σ(weight) ==
        weight[1] + weight[2] + weight[3] + weight[4]

∀ t : RealVectorTriple, i : coordinate
    row(vectorTripleToMatrix(t), i) == t[i]


TDLRSPauxil : trait

  includes GCS

     % Because there are four radars, many quantities appear in fours.
     % These "quad" sorts are axiomatized using the "Quad" trait.

  includes Quad(RealQuad, RealVector, Real)
  includes Array(RealVectorTriple, coordinate, RealVector)
  includes Quad(RealVectorQuad, RealVectorTriple, RealVector)
  includes Quad(NatQuad, NatTriple, Nat)
  includes Quad(BoolQuad, BoolTriple, Bool)

  introduces
     % Each beam's count is converted linearly to a speed,
     % using constant gain and offset.
     beamVelocity : NatQuad → RealQuad
     TDLR_gain, TDLR_offset :→ Real

     % Four vectors of direction cosines in the vehicle coordinates
     % determine the beam directions.
     TDLR_angles :→ RealVectorQuad
```

$cos\_alpha, cos\_beta, cos\_gamma :\rightarrow Real$

**asserts**

**equations**

$TDLR\_angles[1] == [cos\_alpha, cos\_beta, cos\_gamma]$
$TDLR\_angles[2] == [cos\_alpha, -cos\_beta, cos\_gamma]$
$TDLR\_angles[3] == [cos\_alpha, -cos\_beta, -cos\_gamma]$
$TDLR\_angles[4] == [cos\_alpha, cos\_beta, -cos\_gamma]$

$\forall\ nq : NatQuad, b : beam$
$beamVelocity(nq)[b] ==$
$TDLR\_offset + (TDLR\_gain * natToReal(nq[b]))$

### A.3.5 Temperature

```
% Temperature Sensor Processing (TSP)
%
% GCS needs a measurement of temperature in order to correct its
% accelerometer and gyroscope readings.
% The lander has two temperature sensors:
% one solid-state, the other a matched pair of thermocouples.
% The solid-state sensor is accurate over a greater range,
% while the thermocouples are more precise.
% TSP yields the most precise temperature measurement.

TSP : trait

    % The calibration of TSP involves polynomial fitting.
    includes PolynomialFit(Nat, Real, natToReal,
        calibration for pairSet)
    includes DerivativeFit(Nat, natToReal)
    includes GCS

    introduces
        % The temperature depends on reading from both the solid-state (Nat)
        % and thermocouple (Nat) sensors.
        temperature : Nat, Nat → Real                  % most precise temperature
        tempSS : Nat → Real                             % solid-state sensor temp
        tempThermo : Nat → Real                   % thermocouple pair sensor temp
        inRangeThermo : Nat → Bool              % accurate range of thermocouples

        % The solid-state sensor is linear.
        % Its calibration consists of two measurements.
        m1, m2 :→ Nat
        t1, t2 :→ Real
        calibrateSS :→ calibration

        % The thermocouple pair is linear between its calibration points.
        m3, m4 :→ Nat
        t3, t4 :→ Real
        calibrateThermo :→ calibration

        % The thermocouple pair is accurate for readings + or - 15 percent beyond
    % its calibration points. Beyond the calibration points, its response
    % is quadratic. The calibration in the quadratic regions consists of
    % the second, first, and zeroth derivatives.
    min, max :→ Real
    slope :→ Real
    extrapolate :→ Real                                              % 15 percent
    upperCalibration, lowerCalibration :→ RealSequence

asserts
    equations
        calibrateSS == {[$m_1$, $t_1$], [$m_2$, $t_2$]}
        calibrateThermo == {[$m_3$, $t_3$], [$m_4$, $t_4$]}
        $m_3 < m_4$
        min == natToReal($m_3$) − (extrapolate ∗ (natToReal($m_4 \ominus m_3$)))
        max == natToReal($m_4$) + (extrapolate ∗ (natToReal($m_4 \ominus m_3$)))
        slope == ($t_4 - t_3$)/natToReal($m_4 \ominus m_3$)
        upperCalibration == [2, slope, $t_4$]
        lowerCalibration == [−2, slope, $t_3$]

    ∀ nSS, nTh : Nat
        temperature(nSS, nTh) ==
            if inRangeThermo(nTh)
            then tempThermo(nTh)
            else tempSS(nSS)

    ∀ n : Nat
        inRangeThermo(n) == between(min, natToReal(n), max)
        tempSS(n) ==
            value(determinePolynomial(calibrateSS), n)
        tempThermo(n) ==
            if $n > m_4$
            then
            value(determinePolynomial($m_4$, upperCalibration), n)
            else if $n < m_3$
            then
            value(determinePolynomial($m_3$, lowerCalibration), n)
            else
            value(determinePolynomial(calibrateThermo), n)
```

### A.3.6 Guidance

% Guidance Processing (GP)
%
% Computations in GP are needed
% to find acceptably precise values for velocity and altitude if
% the measured values in the current frame are unreliable.
% The measured value of velocity comes from TDLR.
% The measured value of altitude comes from AR.

*GP* : **trait**

**includes** *GCS*
**includes** *Rotation*(*attitude* **for** *integrate*)
**includes** *TDLRSP*
**includes** *ARSP*

**introduces**
% GCS's best estimate of the current velocity depends on TDLR
% (NatQuad and BoolQuad) if that measurement exists, or on
% an integration using velocity, acceleration, and spin
% (RealVectorHistory**3) if it doesn't exist.
*velocity* : *NatQuad*, *BoolQuad*,
*RealVectorHistory*, *RealVectorHistory*, *RealVectorHistory* $\rightarrow$ *RealVector*

% The integration is simply a modification to the previous velocity (Real)
% using the current acceleration (RealVector) and
% the history of angular velocity (RealVectorHistory).
*velocityIntegral* : *RealVector*, *RealVector*, *RealVectorHistory* $\rightarrow$ *RealVector*

% GCS's best estimate of the current altitude depends on ARSP
% (the first four args) if that measurement is reliable, or on
% integration using velocity and spin (the last two args)
% if the measurement isn't reliable.
*altitude* : *Nat*, *Bool*, *RealHistory*, *BoolHistory*,
*RealVectorHistory*, *RealVectorHistory* $\rightarrow$ *Real*

% The integration is a modification to the previous altitude (Real)

```
% using the current velocity (RealVector) and the history of
% angular velocity (RealVectorHistory).
altitudeIntegral : Real, RealVector, RealVectorHistory → Real

% Which way is down depends on the history of rotations.
down : RealVectorHistory → RealVector

% The acceleration due to gravity
gravity :→ Real
```

**asserts**

```
∀ nq : NatQuad, bq : BoolQuad,
    vel, accel, spin : RealVectorHistory, i : coordinate
    velocity(nq, bq, vel, accel, spin)[i] ==
        if velocityOK(nq, bq)[i]
        then velocity(nq, bq)[i]                          % from TDLR
        else velocityIntegral(now(vel), now(accel), spin)[i]

        % A new value for velocity is gotten by
        % 1) using the current spin to change coordinates of previous velocity;
        % 2) adding the measured acceleration in the lander's inertial
        % reference frame (measured in the new vehicle coordinates);
        % 3) adding the acceleration due to gravity of the inertial frame.
∀ vel, accel : RealVector, spin : RealVectorHistory
    velocityIntegral(vel, accel, spin) ==
        (vectorToTensor(delta_t * now(spin)) ⊗ vel)+
        (delta_t * accel)+
        (delta_t * (gravity * down(spin)))

∀ count : Nat, echo : Bool, alt : RealHistory, bh : BoolHistory,
    vel, spin : RealVectorHistory
    altitude(count, echo, alt, bh, vel, spin) ==
        if altitudeOK(echo, bh)
        then altitude(count, echo, alt)                   % from AR
        else altitudeIntegral(now(alt), now(vel), spin)

        % A new value for altitude is gotten by subtracting
        % the downward projection of velocity from the old value.
∀ alt : Real, vel : RealVector, spin : RealVectorHistory
    altitudeIntegral(alt, vel, spin) ==
        $alt - (down(spin) \cdot vel)$

        % "down" is a vector pointing toward the planet.
        % We need it expressed in the lander's coordinates.
        % Assuming the spin history records rotation starting with the lander
        % vertical, then down is the coordinate transformation of the x axis.
$\forall$ $spin$ : $RealVectorHistory$
    $down(spin)$ ==
        $attitude(spin) \otimes unit(x)$
```

## A.4 Actuator Control

## A.4.1 Axial Engines

% Axial Engine Control Law Processing (AECLP)
%
% The lander has three main engines parallel to its vertical axis.
% These engines provide separately controllable thrust to change the
% lander's pitch, yaw, and speed of descent.
%
% AECLP controls the warm-up of the axial engines and, once the engines
% are warm, controls the valve settings that determine engine thrust.
% This control is based on the current data about the lander's dynamics,
% which are in turn computed from sensor data.
%
% AECLP is intended to keep the lander's vertical axis aligned with the
% lander's velocity vector.
% If that is achieved, the axial engines work to slow the lander's descent.
% AECLP is intended to keep the lander's speed close to a pre-computed
% value that depends on the altitude.
% This pre-computed dependence is called the "velocity-altitude contour".
%
% To slow the lander down, more thrust is needed uniformly from all engines.
% To alter pitch or yaw, torque is supplied by differences in the
% thrust from the three axial engines.
%
% Some key sorts used in AECLP are defined in AECLPauxil.
%
% Some AECLP definitions differ from SR's because SR seems to have
% negated certain quantities for unknown reasons.

*AECLP* : **trait**

**includes** *AECLPauxil*

**introduces**
% AEcommand defines a NatTriad of correct valve settings
% based on available data about altitude (RealHistory), velocity,
% acceleration, angular velocity (RealVectorHistory**3),
% the current integrated thrust (Real), and the current phase of descent.
*AEcommand* :
*RealHistory*,
*RealVectorHistory*,

```
RealVectorHistory,
RealVectorHistory,
Real,
phase → NatTriad

% The valve settings are a linear function of the correcting force
% needed to control pitch, yaw, and thrust.
thrust : Real, Real, Real → NatTriad
gp1, gp2, gpy :→ Real                                    % coefficients

% Constants used in defining thrust
TE_init :→ Real                      % thrust while warming engines
TE_drop :→ Real                    % thrust while dropping to contour

% Conversion from real values to valve settings.
convert : Real → Nat
factor :→ Real                                                 % 127

% The "integrated thrust" referred to previously is part of
% the control state updated in trait Frame.
% When the integrated thrust is used in AEcommand to compute the new
% thrust, however, its value is limited by this function:
eThrustL : Real → Real
TE_min, TE_max :→ Real

% Similarly, the restoring torques for pitch and yaw are limited
% by these functions:
ePitchL, eYawL : Real → Real
PE_min, PE_max :→ Real
YE_min, YE_max :→ Real

% Apparently to avoid sudden changes in thrust,
% the computed thrust is integrated.
% Euler integration is explicitly specified;
% to do this we add a component to the GCS state,
% saving the value of the thrust integral in the spec.
% The first argument to eThrust is the result of PID control.
% The second argument is the previous thrust integral.
eThrust : Real, Real → Real
omega, ga :→ Real                                        % coefficients
```

```
asserts
    % AEcommand is determined differently for different phases of the descent.
    %
    ∀ alt : RealHistory, vel, acc, spin : RealVectorHistory, thr : Real, ph : phase
        AEcommand(alt, vel, acc, spin, thr, ph) ==
            if ph.engine = cold ∨ ph.landing
            then 0                                          % all engines off
            else if ph.engine = warming
            then thrust(
            gq * now(spin)[y],
            gr * now(spin)[z],                              % differs from SR
            TE_init)
            else if ph.contour
            then thrust(
            ePitchL(ePitch(currentPitch(spin, vel))),
            eYawL(eYaw(currentYaw(spin, vel))),
            eThrustL(eThrust(
            eVel(currentSpeed(alt, vel, acc)), thr)))
            else thrust(
            ePitchL(ePitch(currentPitch(spin, vel))),
            eYawL(eYaw(currentYaw(spin, vel))),
            TE_drop)

    ∀ ePitch, eYaw, eThrust : Real
        thrust(ePitch, eYaw, eThrust)[1] ==
            convert((gp_1 * ePitch) + eThrust)
        thrust(ePitch, eYaw, eThrust)[2] ==
            convert(((gp_2 * ePitch) - (gpy * eYaw)) + eThrust)
        thrust(ePitch, eYaw, eThrust)[3] ==
            convert(((gp_2 * ePitch) + (gpy * eYaw)) + eThrust)

    ∀ r : Real
        convert(r) == realToNat(factor * cutoff(0, r, 1))

    ∀ r : Real
        eThrustL(r) == cutoff(TE_min, r, TE_max)
        ePitchL(r) == cutoff(PE_min, r, PE_max)
        eYawL(r) == cutoff(YE_min, r, YE_max)
        % Integrate the direct calculation for thrust.
        % Omega is the time constant for the integration.
    ∀ control, thr : Real
      eThrust(control, thr) ==
        ((ga * control * delta_t) + ((1 - (omega * delta_t)) * thr))


AECLPauxil : trait

  includes GCS
  includes Integration(RealHistory, Real, delta_t)

    % The thrust is controlled according to a proportional-integral-derivative
    % (PID) control law. This means the control is a linear function of position
    % in a three dimensional "configuration space" formed by
    % an axis for the error, or deviation from the desired value (P),
    % an axis for integral of the error (I), and
    % an axis for the derivative, or rate of change, of the error (D).
    %
  configuration tuple of
    error : Real,
    integral : Real,
    rate : Real

    % Because there are three engines, many quantities appear in threes.
    % We include traits for arrays of Real and Nat.
  engine enumeration of 1, 2, 3
  includes Array(RealTriad, engine, Real)
  includes Array(NatTriad, engine, Nat)

  introduces
    % The PID control laws each map a configuration into a real number
    % proportional to the correcting force.
    ePitch, eYaw, eVel : configuration → Real

    % Constants used in PID control
    gq, gw, gwi :→ Real                                       % pitch
    gr, gv, gvi :→ Real                                         % yaw
    gax, gve, gvei :→ Real                                   % thrust
```

```
% The current configurations for pitch, yaw, and speed
% are extracted from histories, integrating where necessary.
currentPitch, currentYaw :
RealVectorHistory,
RealVectorHistory → configuration
currentSpeed :
RealHistory,
RealVectorHistory,
RealVectorHistory → configuration

% To express the integrals of pitch and yaw,
% we first form the histories of these quantities.
pitchHistory, yawHistory :
RealVectorHistory → RealHistory

% The speed error is the deviation from a pre-computed
% "velocity-altitude contour".
% GCS tries to follow this contour.
% speedError depends on the histories of altitude and velocity.
speedError :
RealHistory,
RealVectorHistory → RealHistory

% The precomputed speed that the lander should have at a given altitude.
velocityAltitudeContour : Real → Real
```

**asserts**

```
    ∀ pitch : configuration
        ePitch(pitch) ==
            (gq * pitch.rate)+
            (gw * pitch.error)+
            (gwi * pitch.integral)

    ∀ yaw : configuration
        eYaw(yaw) ==
            (gr * yaw.rate)+                              % differs from SR
            (gv * yaw.error)+
            (gvi * yaw.integral)
```

```
∀ velError : configuration
    eVel(velError) ==
        ((−gax) ∗ velError.rate)+
        (gve ∗ velError.error)+
        (gvei ∗ velError.integral)

        % The history of pitch and yaw angles is gotten from the
        % 3-dimensional history of velocity
        % expressed in the lander's coordinates.
        % Because we only care about differences between the velocity vector
        % and the lander's x-axis (vertical),
        % pitch is the angle between them in the x-z plane, and
        % yaw is the angle between them in the x-y plane.
        % The small-angle approximation is used in this specification,
        % which is an optimistic assumption about the control.
∀ velH : RealVectorHistory, i : Nat
    pitchHistory(velH)[i] == (velH[i])[z]/(velH[i])[x]
    yawHistory(velH)[i] == −((velH[i])[y]/(velH[i])[x])
        % definition of yaw differs from SR

        % The angular velocity measurement provides the derivatives of
        % pitch and yaw.
∀ spin, vel : RealVectorHistory
    currentPitch(spin, vel) ==
        [now(pitchHistory(vel)),
        integrate(pitchHistory(vel)),
        now(spin)[y]]
    currentYaw(spin, vel) ==
        [now(yawHistory(vel)),
        integrate(yawHistory(vel)),
        now(spin)[z]]

        % The speed error is the difference from the precomputed speed
        % the lander should have.
∀ altH : RealHistory, velH : RealVectorHistory, i : Nat
    speedError(altH, velH)[i] ==
        length(velH[i]) − velocityAltitudeContour(altH[i])

        % The acceleration measurement provides the derivative of
        % speedError. This is an approximation that assumes:
        % 1) changes in the velocity-altitude contour are small compared to
        % the acceleration;
        % 2) the components of velocity and acceleration along the lander's
        % vertical axis are much larger than the other components.
    ∀ alt : RealHistory, vel, acc : RealVectorHistory
      currentSpeed(alt, vel, acc) ==
        [now(speedError(alt, vel)),
        integrate(speedError(alt, vel)),
        now(acc)[x]]
```

## A.4.2 Roll Engines

```
% Roll Engine Control Law Processing (RECLP)
%
% The lander has three opposing pairs of roll engines mounted on its sides
% perpendicular to the vertical axis.
% Firing these engines provides torque around that axis, and can be used
% to control roll.
% RECLP attempts to maintain the roll angle at a fixed reference point
% determined at initialization.
% (This reference point seems to have no physical significance, because
% the lander is simultaneously rotating around the other axes too.
% However, if the rotation is not too violent, fixing the roll angle
% may be sufficient stability for controlling the axial engines.)

RECLP : trait

  includes RECLPauxil
  includes Integration(RealHistory, Real, delta_t)

  introduces
    % The roll engines will be fired with an intensity and direction
    % determined by the history of vehicle rotation (RealVectorHistory),
    % and the current phase of the descent.
    % (Do not confuse the phase of descent with phase space.)
    REintensity : RealVectorHistory, phase → REintensity
    REsense : RealVectorHistory → sense
```

```
% The roll rate is simply the x component of the spin vector.
% The roll angle is gotten by integrating the rate.
% The combination of angle and rate forms a phase space for roll.
currentPhasePosition : RealVectorHistory → rollPhaseSpace

% The roll rate is one of the components of spin, and
% so can be projected out.
rollHistory : RealVectorHistory → RealHistory


asserts
  ∀ spin : RealVectorHistory, ph : phase
    REintensity(spin, ph) ==
      if ph.engine = cold ∨ ph.landing
      then off
      else REpulse(currentPhasePosition(spin))
    REsense(spin) ==
      REdirection(currentPhasePosition(spin))

  ∀ spin : RealVectorHistory, n : Nat
    currentPhasePosition(spin) ==
      [integrate(rollHistory(spin)),
      now(spin)[x]]

    rollHistory(spin)[n] == (spin[n])[x]
```

```
RECLPauxil : trait

    includes GCS

    REintensity enumeration of off, minimum, intermediate, maximum

    rollPhaseSpace tuple of
        angle : Real,
        rate : Real

    introduces
        % The intensity and direction depend only on position in the
        % roll phase space. The phase space is divided into rectangular
        % regions using some constants as boundaries.
        % (This seems unnecessarily ad hoc to me; why not use a PD control law
        % similar to those for pitch and yaw in AECLP? The thrust
        % determined by this control law would need to be quantized in that case,
        % of course, but the same is true for AECLP.)
        REpulse : rollPhaseSpace → REintensity
        REdirection : rollPhaseSpace → sense
        theta_1, theta_2 :→ Real
        p_1, p_2, p_3, p_4 :→ Real

    asserts
        ∀ roll : rollPhaseSpace
            REdirection(roll) ==
                if (roll.rate ≥ p_4)∨
                (roll.rate ≥ p_1 ∧ between(0, roll.angle, π))∨
                (roll.rate ≥ 0 ∧ between(theta_1, roll.angle, π))∨
                (roll.rate ≥ (−p_3) ∧ between(theta_2, roll.angle, π))
                then clockWise
                else counterClockWise

            REpulse(roll) ==
                if roll.rate < 0
                then
                REpulse([−roll.angle, −roll.rate])
                else
                if (between(p_3, roll.rate, p_4)∧
                between(−π, roll.angle, 0))∨
```

```
$(between(0, roll.rate, p_4) \wedge$
$between(-theta_2, roll.angle, 0)) \vee$
$(between(0, roll.rate, p_1) \wedge$
$between(0, roll.angle, theta_1))$
then $off$
else
if $between(0, roll.rate, p_1) \wedge$
$between(theta_1, roll.angle, theta_2)$
then $minimum$
else
if $between(p_1. roll.rate, p_2) \wedge$
$between(0, roll.angle, theta_2)$
then $intermediate$
else $maximum$
```

## A.5 I/O Interface

% This trait defines the connection between the sorts used throughout
% the rest of the specification and the sorts that model details of
% input and output registers used by GCS.

*Interface* : **trait**

    **includes** *Frame*
    **includes** *Communication*

        % 8- 16- and 32-bit registers are used in GCS I/O.
        %
    **includes** *Register*(8, *8bit* **for** *register*)

    **includes** *Register*(16, *16bit* **for** *register*)
    **includes** *Map*(*16bitTriple*, *coordinate*, *16bit*)
    **includes** *Map*(*16bitTriad*, *engine*, *16bit*)
    **includes** *Map*(*16bitQuad*, *beam*, *16bit*)

    **includes** *Register*(32, *32bit* **for** *register*)
    **includes** *Map*(*32bitQuad*, *beam*, *32bit*)

        % Define the concrete representation of the sensor data sort
        % used in trait Frame. All registers are 16 bit except the counters
        % used to keep TDLR unlocked for an interval of time.
        %
    *input* **tuple of**
        *AR_counter* : *16bit*,
        *A_counter* : *16bitTriple*,
        *G_counter* : *16bitTriple*,
        *TDLR_counter* : *16bitQuad*,
        *SS_temp* : *16bit*,
        *Thermo_temp* : *16bit*,
        *TD_counter* : *16bit*,

        *TDLR_unlocked* : *32bitQuad*

        % Define the concrete representation of the actuator data sort

```
    % used in trait Frame.
    %
output tuple of
    AE_cmd : 16bitTriad,
    RE_cmd : 16bit,
    chute_released : Bool

    % Define the concrete representation of communication packets
    % used in trait Communication.
    %
commOutput tuple of
    pattern : 16bit,
    number : 8bit,
    variables : bitSequence,
    checksum : 16bit

introduces
    % Abstraction functions map the concrete sorts into the abstract.
    abstract : input → sensor
    abstract : output → actuator
    abstract : commOutput → packet

    % There is only one instance of concrete I/O state being preserved
    % from one frame to the next. This instance is in TDLRSP, where
    % counters record how long until each radar beam can be locked again.
    % The update function tells how this concrete I/O state is modified
    % between frames.
    update : input → input
    TDLR_lock_time :→ Real                                          % a constant

    % Roll engine intensities are coded in 2 bits.
    % This function decodes those bits.
    REconvert : Nat → REintensity

    % The GCS programmer must choose a set of variables to be communicated
    % and a packing of these variables into a sequence of bits.
    % This function unpacks those bits.
    % We do not specify any properties of the packing.
    unpack : bitSequence → setOfVariables
```

```
asserts
    % If the altimeter input is not all ones,
    % take the value in the lowest 10 bits.
    $\forall\ in : input$
        $abstract(in).AR\_counter\_OK ==$
            $\neg allOnes(in.AR\_counter)$

        $\neg allOnes(in.AR\_counter) \Rightarrow$
        $abstract(in).AR\_counter = value(in.AR\_counter, [0, 10])$

        % Use all 16 bits to get an accelerometer value.
    $\forall\ in : input, i : coordinate$
        $abstract(in).A\_counter[i] == value(in.A\_counter[i])$

            % Bit 15 is the sign; use lowest 14 bits for gyroscope value.
    $\forall\ in : input, i : coordinate$
        $abstract(in).G\_counter[i] ==$
            $value(in.G\_counter[i], [0, 14])$

        $abstract(in).G\_counter\_sense[i] ==$
            if $(in.G\_counter[i])[15]$
            then $clockWise$
            else $counterClockWise$

            % The TDLR value is the full 16 bits in the register when
            % the value is not zero and the _unlocked counter has reached zero.
    $\forall\ in : input, b : beam$
        $abstract(in).TDLR\_counter\_OK[b] ==$
            $(allZeros(in.TDLR\_unlocked[b])) \wedge$
            $value(in.TDLR\_counter[b]) \neq 0$

        $abstract(in).TDLR\_counter\_OK[b] \Rightarrow$
        $abstract(in).TDLR\_counter[b] = value(in.TDLR\_counter[b])$

        % Use all 16 bits for each temperature sensor.
    $\forall\ in : input$
        $abstract(in).SS\_temp == value(in.SS\_temp)$
        $abstract(in).Thermo\_temp == value(in.Thermo\_temp)$
```

% The touch-down input should be either all ones or all zeros.
$\forall\ in : input$
$allOnes(in.TD\_counter) \Rightarrow abstract(in).touch\_down$
$allZeros(in.TD\_counter) \Rightarrow \neg abstract(in).touch\_down$

% Each TDLR radar may become "unlocked", after which its
% value is not reliable for a time.
% The TDLR_unlocked counter prevents GCS from using the value
% returned by a TDLR beam until that time has passed.
% Decrement the TDLR_unlocked counter between frames unless
% it is already zero and the beam is returning non-zero values.
% If the counter has been zero but the beam returns zero,
% reset the counter to the required wait.
$\forall\ in : input, b : beam$
$value(in.TDLR\_unlocked[b]) > 0 \Rightarrow$
$value(update(in).TDLR\_unlocked[b]) =$
$value(in.TDLR\_unlocked[b]) \ominus 1$

$(value(in.TDLR\_unlocked[b]) = 0 \wedge$
$value(in.TDLR\_counter[b]) = 0) \Rightarrow$
$value(update(in).TDLR\_unlocked[b]) =$
$realToNat(TDLR\_lock\_time / delta\_t)$

$(value(in.TDLR\_unlocked[b]) = 0 \wedge$
$value(in.TDLR\_counter[b]) > 0) \Rightarrow$
$allZeros(update(in).TDLR\_unlocked[b])$

% All 16 bits matter for each axial engine.
$\forall\ out : output, e : engine$
$abstract(out).AE\_cmd[e] == value(out.AE\_cmd[e])$

% Show the meaning of the two intensity bits in RE.
$\forall\ n : Nat$
$n = 0 \Rightarrow REconvert(n) = off$
$n = 1 \Rightarrow REconvert(n) = minimum$
$n = 2 \Rightarrow REconvert(n) = intermediate$
$n = 3 \Rightarrow REconvert(n) = maximum$

% Bits 1 and 2 hold the intensity: bit 0 the sense.
$\forall\ out : output$

```
        abstract(out).RE_cmd_intensity ==
            REconvert(value(out.RE_cmd, [1, 2]))

        abstract(out).RE_cmd_sense ==
            if out.RE_cmd[0]
            then clockWise
            else counterClockWise

    ∀ out : output
        abstract(out).release_chute == out.chute_released

            % Define the packet corresponding to concrete communication output.
    ∀ comm : commOutput
        abstract(comm).pattern ==
            value(comm.pattern)
        abstract(comm).number ==
            value(comm.number)
        abstract(comm).internal ==
            unpack(comm.variables)
        abstract(comm).checksum ==
            value(comm.checksum)
```

## A.6 Auxiliary Traits

### A.6.1 Quads

```
% The Quad trait is used in TDLRSP.
% A quad consists of four objects of the same sort,
% corresponding to the four touch down landing radar beams.

Quad(quadSort, tripleSort, range) : trait

    beam enumeration of 1, 2, 3, 4
    includes Map(quadSort, beam, range)

    coordinate enumeration of x, y, z
    assumes Map(tripleSort, coordinate, range)

        % The only new operator we define for quads simply throws away one
        % of the four components to yield a triple.
        % There are four ways to do this, while preserving the original quad order.
        %
    introduces
        ignore : beam, quadSort → tripleSort

    asserts
        ∀ q : quadSort
            ignore(1, q)[x] == q[2]
            ignore(1, q)[y] == q[3]
            ignore(1, q)[z] == q[4]

            ignore(2, q)[x] == q[1]
            ignore(2, q)[y] == q[3]
            ignore(2, q)[z] == q[4]

            ignore(3, q)[x] == q[1]
            ignore(3, q)[y] == q[2]
            ignore(3, q)[z] == q[4]

            ignore(4, q)[x] == q[1]
            ignore(4, q)[y] == q[2]
            ignore(4, q)[z] == q[3]
```

### A.6.2 Rotations

% Rotations are RealTensors that preserve vector length.
% This property of tensors is called
*orthogonality.*
%
% In three dimensions a unique but ersatz "vector" can be
% associated with each rotation tensor.
% According to Euler's theorem, each three dimensional rotation has an axis;
% the rotation vector lies along this axis and has magnitude proportional to
% the rotation angle.
% Rotation "vectors" are ersatz because composition of rotations is not
% modeled by vector sum, but by tensor product.
%
% This trait axiomatizes the relation between rotation "vectors" and tensors,
% and defines the integral of a history of rotations.
% We will need this trait to describe rotations of the lander, so
% we define a rotation as a coordinate transformation,
% rather than a change in vectors.

*rotation* : **trait**

**includes** *Real*
**includes** *Vector*(*RealVector*, *Real*)
**includes** *Tensor*(*RealTensor*, *Real*, *RealVector* **for** *vector*)
**includes** *History*(*RealVectorHistory*, *RealVector*)

**introduces**
% conversions between rotation vector and tensor
*vectorToTensor* : *RealVector* → *RealTensor*
*tensorToVector* : *RealTensor* → *RealVector*

*orthogonal* : *RealTensor* → *Bool*

% Given a sequence of rotation rates specified as vectors,
% where the rates are measured delta apart,
% compose the history of rotations to form a tensor.
*integrate* : *RealVectorHistory* → *RealTensor*

```
delta :→ Real

% vector length and angle between two vectors.
length : RealVector → Real
angle : RealVector, RealVector → Real

% vectors a,b,c are right handed if c is perpendicular to the plane
% formed by a and b, and a screw turned from a to b advances in
% the direction of c.
rightHanded : RealVector, RealVector, RealVector → Bool

asserts
    % orthogonal tensors preserve length.
    ∀ t : RealTensor, v : RealVector
        orthogonal(t) ⇒
        length(t ⊗ v) = length(v)

        % axiomatize vectorToTensor
    ∀ u, v, w : RealVector
        orthogonal(vectorToTensor(v))
        vectorToTensor(0) == 1
        vectorToTensor(v) ⊗ v == v

        (rightHanded(w, u, v)∧
        angle(w, u) = length(v)) ⇒
        (vectorToTensor(v) ⊗ u) = w                          % coordinate change

        vectorToTensor(v) ==
            vectorToTensor(v + (((2 * π)/length(v)) * v))
        vectorToTensor(v) ⊗ u = w ==
            vectorToTensor(−v) ⊗ w = u

    ∀ t : RealTensor
        orthogonal(t) ⇒
        vectorToTensor(tensorToVector(t)) = t

        % Define integration of a sequence of rotations.
    ∀ h : RealVectorHistory, e : RealVector
        integrate(update(h, e)) ==
```

$(delta * vectorToTensor(e)) \otimes integrate(h)$
$integrate(empty) == 1$

% Define length and angle.
$\forall\ v, w : RealVector$
$length(v) == root(2, v \cdot v)$
$v \cdot w == length(v) * length(w) * cosine(angle(v, w))$
$0 \leq angle(v, w)$
$angle(v, w) \leq \pi$

% Axiomatize rightHanded.
% A continuity property is also needed to define it completely.
$\forall\ x, y, z : RealVector$
$rightHanded(x, y, z) \Rightarrow$
$(angle(x, z) = \pi/2 \wedge$
$angle(y, z) = \pi/2 \wedge$
$length(x) > 0 \wedge$
$length(y) > 0 \wedge$
$length(z) > 0)$

$(angle(x, z) = \pi/2 \wedge$
$angle(y, z) = \pi/2 \wedge$
$length(x) > 0 \wedge$
$length(y) > 0 \wedge$
$length(z) > 0) \Rightarrow$
$(rightHanded(x, y, z) \vee rightHanded(x, y, -z))$

**implies**
**converts** *vectorToTensor* % given linearity of tensors

### A.6.3 Vectors and Tensors

% A vector is a magnitude and a direction.
% In this trait we represent a vector as an array in the lander's coordinates.

*Vector*(*vector*, *value*) : **trait**

*coordinate* **enumeration of** $x, y, z$
**includes** *Array*(*vector*, *coordinate*, *value*)
**assumes** *Field*(*value*, $+$, $*$)

**introduces**
$[\_\_, \_\_, \_\_]$ : *value*, *value*, *value* $\rightarrow$ *vector*

**asserts**
$\forall\ a, b, c$ : *value*
$([a, b, c])[x] == a$
$([a, b, c])[y] == b$
$([a, b, c])[z] == c$
$\forall\ v$ : *vector*
$[v[x], v[y], v[z]] == v$
$\forall\ v$ : *vector*
$\Sigma(v) == v[x] + v[y] + v[z]$ % introduced in Array

% A tensor is an object that transforms vectors.
% The transformation is invariant under coordinate change.
% In this trait we represent a tensor as a matrix in the lander's coordinates.
% Vector transformations are represented by matrix multiplication.

*Tensor*(*tensor*, *value*) : **trait**

*coordinate* **enumeration of** $x, y, z$
**assumes** *Vector*(*vector*, *value*)
**includes** *Matrix*(*tensor*, *coordinate*, *value*, *vector* **for** *array*)

**introduces**
$[\_\_, \_\_, \_\_]$ : *vector*, *vector*, *vector* $\rightarrow$ *tensor*

**asserts**
$\forall\ a, b, c : vector$
$row([a, b, c], x) == a$
$row([a, b, c], y) == b$
$row([a, b, c], z) == c$
$\forall\ t : tensor$
$[row(t, x), row(t, y), row(t, z)] == t$

### A.6.4 Integration

% Euler integration over histories of events.
% Assume that the events are samples separated in time by delta.

*Integration*(*history*, *event*, *delta*) : **trait**

**assumes** *History*(*history*, *event*)
**assumes** *Field*(*event*, +, ∗)

**introduces**
% Declare delta to be of the same sort as the events.
$delta : \rightarrow event$

$integrate : history \rightarrow event$

**asserts**
% The Euler method of approximating an integral is just to sum
% the contributions for each (time) step.
$\forall\ h : history, e : event$
$integrate(update(h, e)) == integrate(h) + (delta * e)$
$integrate(empty) == 0$

## A.7 Generic Traits

### A.7.1 Polynomials and Polynomial Fitting

```
% A polynomial is a map from natural numbers to some range sort,
% which must form a ring.
% This map determines the polynomial coefficients.
% Evaluating a polynomial takes some domain into the polynomial's range.
% This mapping is defined in terms of +, *, and ^ for the range sort,
% so we require that there be an operator taking the domain into the range.
```

*Polynomial(domain, range, domainToRange)* : **trait**

  **assumes** *Ring(range, +, *)*
  **includes** *Natural*
  **includes** *TotalOrder(Nat)*

  **introduces**
    *domainToRange* : *domain* → *range*

    *coefficient* : *polynomial, Nat* → *range*
    *degree* : *polynomial* → *Nat*
    *value* : *polynomial, domain* → *range*

    % value of polynomial terms of degree < = Nat
    *value* : *polynomial, domain, Nat* → *range*

    % exponentiation for the range sort
    __^__ : *range, Nat* → *range*

  **asserts**
    *polynomial* **partitioned by** *coefficient*

    $\forall\ p : polynomial, d : domain, n : Nat$
      $n > degree(p) \Rightarrow coefficient(p, n) = 0$
      $coefficient(p, degree(p)) \neq 0$
      $value(p, d) == value(p, d, degree(p))$
      $value(p, d, 0) == coefficient(p, 0)$
      $value(p, d, succ(n)) ==$
        $value(p, d, n) +$
        $(coefficient(p, succ(n)) * (domainToRange(d) \hat{}\, succ(n)))$

% A set of n domain-range pairs determines a unique polynomial
% of degree n-1, given that no value of the domain appears twice in the set.
% We also introduce some convenient notation for sets of pairs.

*PolynomialFit(domain, range, domainToRange)* : **trait**

**includes** *Polynomial(domain, range, domainToRange)*
**assumes** *Field(range, +, *)*
*pair* **tuple of**
  *preimage : domain,*
  *image : range*
**includes** *Set(pair, pairSet, Nat* **for** *Card)*

**introduces**
  $determinePolynomial : pairSet \rightarrow polynomial$
  $inconsistent : pairSet \rightarrow Bool$

  $\{\_\_, \_\_\} : pair, pair \rightarrow pairSet$
  $\{\_\_, \_\_, \_\_\} : pair, pair, pair \rightarrow pairSet$
  $\{\_\_, \_\_, \_\_, \_\_\} : pair, pair, pair, pair \rightarrow pairSet$

**asserts**
  $\forall\ s : pairSet, p : pair$
    $degree(determinePolynomial(s)) == size(s) \ominus 1$

    $(p \in s \wedge$
    $\neg inconsistent(s)) \Rightarrow$
    $(value(determinePolynomial(s), p.preimage) = p.image)$

  $\forall\ s : pairSet, p, q : pair$
    $(p \in s \wedge q \in s \wedge$
    $p.preimage = q.preimage) \Rightarrow$
    $inconsistent(s)$

$\forall\ w, x, y, z : pair$

$\{w, x\} == insert(\{w\}, x)$

$\{w, x, y\} == insert(\{w, x\}, y)$

$\{w, x, y, z\} == insert(\{w, x, y\}, z)$

% A polynomial from reals to reals is differentiable.
% In that case, the values of the zeroth thru nth derivatives at a point
% in the domain will determine a polynomial of degree n.

*DerivativeFit(domain, domainToReal)* : **trait**

**includes** *Polynomial(domain, Real, domainToReal)*
**includes** *Real*
**includes** *Sequence(RealSequence, Real)*

**introduces**
% The RealSequence specifies the derivatives at a point in the domain.
% The zeroth derivative is the newest element of the sequence.
*determinePolynomial* : *domain, RealSequence* $\rightarrow$ *polynomial*

*derivative* : *polynomial* $\rightarrow$ *polynomial*
*derivative* : *Nat, polynomial* $\rightarrow$ *polynomial*

**asserts**
$\forall\ x : domain, s : RealSequence, n : Nat$

$degree(determinePolynomial(x, s)) ==$
$length(s) \dot{-} 1$

$n < length(s) \Rightarrow$
$value(derivative(n, determinePolynomial(x, s)), x) = s[n]$

$\forall\ p : polynomial, n : Nat$

$coefficient(derivative(p), n) ==$
$natToReal(succ(n)) * coefficient(p, succ(n))$

$derivative(succ(n), p) ==$
$derivative(derivative(n, p))$

$derivative(0, p) == p$

## A.7.2 Sequences and Histories

% A history is a sequence of events.

*History*(*history*, *event*) : **trait**

```
includes Sequence(history, event,
    now for head,
    before for tail,
    update for cons,
    previous for shift)
```

% Sequences are constructed by appending elements to the end of
% other sequences. The most recent element is called the head,
% the rest is called the tail.

*Sequence*(*sequence*, *element*) : **trait**

**includes** *Natural*

**introduces**
$cons : sequence, element \rightarrow sequence$
$head : sequence \rightarrow element$
$tail : sequence \rightarrow sequence$

$empty : \rightarrow sequence$
$isEmpty : sequence \rightarrow Bool$
$[] : \rightarrow sequence$
$[\_\_] : element \rightarrow sequence$
$[\_\_, \_\_] : element, element \rightarrow sequence$
$[\_\_, \_\_, \_\_] : element, element, element \rightarrow sequence$

$shift : sequence, Nat \rightarrow sequence$
$\_\_[\_\_] : sequence, Nat \rightarrow element$
$length : sequence \rightarrow Nat$

**asserts**
*sequence* **generated by** *empty*, *cons*
$\forall\ s : sequence, e : element$
$head(cons(s, e)) == e$
$tail(cons(s, e)) == s$
$cons(tail(s), head(s)) == s$

$\forall\ s : sequence$
$isEmpty(s) == s = empty$

$\forall\ x, y, z : element$
$[x] == cons([], x)$
$[x, y] == cons([x], y)$
$[x, y, z] == cons([x, y], z)$

$\forall$ *s* : ***sequence***, ***n*** : *Nat*
  ***shift***$(s, 0) == s$
  ***shift***$(s, succ(n)) ==$ ***shift***$(tail(s), n)$
  $s[n] == head(shift(s, n))$

$\forall$ *s* : ***sequence***, ***e*** : ***element***
  *length*$(empty) == 0$
  *length*$(cons(s, e)) ==$ *length*$(s) + 1$

**implies**
  **converts** *shift*, __[__], *length*

## A.7.3 Linear Equations

% This trait axiomatizes sets of n linear equations in n unknowns.
% These equations are of the form
$c = m \otimes a$,
% where c is a given array, m is a given matrix, and
% a is an array of the unknowns.
% The number of equations and unknowns must be equal because we are
% relying on the theory of square matrices, in which the number of
% rows and columns must be equal.

*LinearEquation*(*array*, *matrix*, *index*, *value*) : **trait**

  **assumes** *Array*(*array*, *index*, *value*)
  **assumes** *Matrix*(*matrix*, *index*, *value*)
  **assumes** *Map*(*BoolArray*, *index*, *Bool*)
  **includes** *Set*(*array*, *arraySet*)
  **includes** *Set*(*value*, *valueSet*)

    % Typically, n linear equations determine a unique solution for n unknowns.
    % However, this is not true in general.
    % Nor will we define an algorithm for determining the solutions.

```
    % Rather, we describe the properties every set of solutions must have,
    % and the dependence of each unknown on the input array c.
    %
introduces

    % solutions and sets of solutions.
    solutions : array, matrix → arraySet
    solutions : arraySet, matrix → arraySet
    solve : array, matrix → array

    % If BoolArray indicates which components of input array c are fixed,
    % determine indicates which components of unknown array a are determined.
    determine : BoolArray, array, matrix → BoolArray

    % arrays match at components where BoolArray is true.
    match : BoolArray, array, array → Bool

    % set of arrays matching at BoolArray.
    matches : BoolArray, array → arraySet

    % the set of values of a given array component
    __[__] : arraySet, index → valueSet

    % uniqueness for each array component
    unique : arraySet → BoolArray

asserts
    ∀ m : matrix, a, c : array, s : arraySet
        a ∈ solutions(c, m) == m ⊗ a = c
        a ∈ solutions(s, m) == (m ⊗ a) ∈ s
        solve(c, m) ∈ solutions(c, m)

    ∀ b : BoolArray, c : array, m : matrix
        determine(b, c, m) ==
            unique(solutions(matches(b, c), m))

    ∀ b : BoolArray, a, c : array, i : index
        match(b, a, c) ==
```

$b[i] \Rightarrow a[i] = c[i]$
$a \in matches(b, c) ==$
$match(b, a, c)$

$\forall\ s : arraySet, a : array, i : index$
$(\{\})[i] = \{\}$
$insert(s, a)[i] == insert(s[i], a[i])$

$\forall\ s : arraySet, i : index$
$unique(s)[i] == size(s[i]) = 1$

## A.7.4 Arrays and Matrices

% An array is a map from an index sort to a value sort,
% where the value sort is a field.
% The field requirement allows us to describe vector-like operations,
% such as inner product.
% However, arrays are not vectors, because coordinates and coordinate
% transformations have not been specified.

*Array(array, index, value)* : **trait**

**includes** *Map(array, index, value)*
**assumes** *Field(value, +, *)*

**introduces**

| | |
|---|---|
| $\Sigma : array \rightarrow value$ | % sum of array components |
| $\_\_ + \_\_ : array, array \rightarrow array$ | % component-wise sum |
| $\_\_ * \_\_ : array, array \rightarrow array$ | % component-wise product |
| $\_\_ \cdot \_\_ : array, array \rightarrow value$ | % inner product |
| $\_\_ * \_\_ : value, array \rightarrow array$ | % left scalar product |
| $\_\_ * \_\_ : array, value \rightarrow array$ | % right scalar product |
| $- \_\_ : array \rightarrow array$ | % negation |
| $unit : index \rightarrow array$ | % unit vector |
| $0 : \rightarrow array$ | % a constant |

**asserts**

$\forall\ a_1, a_2 : \mathit{array}, i : \mathit{index}$
- $(a_1 + a_2)[i] == a_1[i] + a_2[i]$
- $(a_1 * a_2)[i] == a_1[i] * a_2[i]$
- $a_1 \cdot a_2 == \Sigma(a_1 * a_2)$

$\forall\ c : \mathit{value}, a : \mathit{array}, i : \mathit{index}$
- $(c * a)[i] == c * a[i]$
- $(a * c)[i] == c * a[i]$
- $-a == (-1) * a$

$\forall\ i, j : \mathit{index}$
- $\mathit{unit}(i)[j] ==$
  **if** $i = j$ **then** 1 **else** 0
- $0[j] == 0$

```
% The sigma operator is not converted because nothing is yet specified
% about the index sort to be summed over.
%
```

**implies**

**converts** $\_\_ + \_\_ : \mathit{array}, \mathit{array} \rightarrow \mathit{array}$,
$\_\_ * \_\_ : \mathit{array}, \mathit{array} \rightarrow \mathit{array}$,
$\_\_ * \_\_ : \mathit{value}, \mathit{array} \rightarrow \mathit{array}$,
$\_\_ * \_\_ : \mathit{array}, \mathit{value} \rightarrow \mathit{array}$,
$-\_\_ : \mathit{array} \rightarrow \mathit{array}$,
$\cdot$, $\mathit{unit}$,
$0 : \rightarrow \mathit{array}$

```
% A matrix is a map from an element, which is a pair of indices,
% to a value, where the possible values form a field.
% Because we have used the same index sort for both components of the pair,
% we are axiomatizing square matrices only.

Matrix(matrix, index, value) : trait

  assumes Array(array, index, value)
  element tuple of row, column : index
  includes Map(matrix, element, value)

  introduces
     row : matrix, index → array                          % index by row arrays
     column : matrix, index → array                    % index by column arrays
     __ + __ : matrix, matrix → matrix                    % component-wise sum
     __ * __ : value, matrix → matrix                      % left scalar product
     __ * __ : matrix, value → matrix                     % right scalar product
     __ ⊙ __ : matrix, matrix → matrix                          % multiplication
     __ ⊙ __ : matrix, array → array

                                                         % right multiplication
     __ ⊗ __ : array, matrix → array

                                                          % left multiplication
     0, 1 :→ matrix                                                 % constants

  asserts
     ∀ m, m1, m2 : matrix, a : array, e : element, i : index, c : value
        m[e] == row(m, e.row)[e.column]
        m[e] == column(m, e.column)[e.row]
        (m1 + m2)[e] == m1[e] + m2[e]
        (m1 ⊙ m2)[e] == row(m1, e.row)·
           column(m2, e.column)
        (c * m)[e] == c * m[e]
        (m * c)[e] == c * m[e]
        (m ⊙ a)[i] == row(m, i) · a
        (a ⊗ m)[i] == a · column(m, i)
        0[e] == 0
        1[e] == if e.row = e.column then 1 else 0

  implies
     Field(matrix, +, ⊙)
```

```
converts row, column,
    0 :→ matrix,
    1 :→ matrix,
    __ + __ : matrix, matrix → matrix,
    __ * __ : matrix, value → matrix,
    __ * __ : value, matrix → matrix,
    __ ⊗ __ : matrix, array → array,
    __ ⊗ __ : array, matrix → array,
    __ ⊗ __ : matrix, matrix → matrix
```

### A.7.5 Registers

```
% A register is a finite sequence of bits.
% It is often interpreted as a natural number value, signed or unsigned.
% Note that bits in a register are numbered here starting with 0.

Register(width) : trait

  includes Map(register, Nat, Bool)
  includes Natural

  interval tuple of
      base : Nat,
      offset : Nat

  introduces
      width :→ Nat                         % the number of bits in the register

      % Define the unsigned value of a register,
      % and of a interval of consecutive bits within the register.
      value : register → Nat
      value : register, interval → Nat

      % auxiliary functions.
      inInterval : Nat, interval → Bool
      allZeros : register → Bool
      allOnes : register → Bool
```

**asserts**

$\forall\ r : register, n, m : Nat$

$value(r) == value(r, [0, width])$

$value(r, [n, succ(m)]) ==$

$(\ \textbf{if}\ r[n + m]\ \textbf{then}\ 2\hat{}m\ \textbf{else}\ 0) +$

$value(r, [n, m])$

$value(r, [n, 0]) == 0$

$\forall\ n : Nat, i : interval$

$inInterval(n, i) == i.base \leq n \wedge (n < (i.base + i.offset))$

$\forall\ r : register, n : Nat$

$inInterval(n, [0, width]) \Rightarrow$

$(allZeros(r) \Rightarrow \neg r[n])$

$inInterval(n, [0, width]) \Rightarrow$

$(allOnes(r) \Rightarrow r[n])$

## A.7.6 Mappings

% The GCS specification has many sorts that map one sort into another.
% In each case, the following trait will be included.

*Map(map, domain, range)* : **trait**

**introduces**

$\_\_[\_\_] : map, domain \rightarrow range$

**asserts**

*map* **partitioned by** $\_\_[\_\_]$

C-2

## A.7.7 Statistics

% The statistics trait defines statistical properties of collections of Reals.
% These properties are used in the ASP trait.
% We also introduce some convenient notation for bags of Reals.

*Statistics* : **trait**

**includes** *Real*
**includes** *Bag*(*Real*, *RealBag*, *Nat* **for** *Card*)

**introduces**

| | |
|---|---|
| $sum : RealBag \rightarrow Real$ | % sum of bag elements |
| $mean : RealBag \rightarrow Real$ | % mean of bag elements |
| $varianceSum : RealBag \rightarrow Real$ | % sum of squared deviations |
| $variance : RealBag \rightarrow Real$ | % mean-square |
| $standardDeviation : RealBag \rightarrow Real$ | % root-mean-square |

$\{\_\_, \_\_\} : Real, Real \rightarrow RealBag$
$\{\_\_, \_\_, \_\_\} : Real, Real, Real \rightarrow RealBag$
$\{\_\_, \_\_, \_\_, \_\_\} : Real, Real, Real, Real \rightarrow RealBag$

**asserts**
$\forall\ b : RealBag, r : Real$
$sum(\{\}) == 0$
$sum(insert(b, r)) == r + sum(b)$
$mean(b) == sum(b)/natToReal(size(b))$

% This definition of variance is slightly different than the usual
% (for a finite sample, should use size(b)-1 rather than size(b)),
% but it is the one required in SR.
$varianceSum(\{\}) == 0$
$varianceSum(insert(b, r)) ==$
$((r - mean(b))\hat{}2) + varianceSum(b)$
$variance(b) = varianceSum(b)/natToReal(size(b))$
$standardDeviation(b) == root(2, variance(b))$

$\forall\ w, x, y, z : Real$
$\{w, x\} == insert(\{w\}, x)$
$\{w, x, y\} == insert(\{w, x\}, y)$

$\{w, x, y, z\} == insert(\{w, x, y\}, z)$

### A.7.8 Real Numbers

```
% Everything we need to know about real numbers for GCS.
%
% The reals form a totally ordered field.
% We introduce some additional operators, including division and
% extraction of roots.
% We also axiomatize conversions to and from the natural numbers.
```

*Real* : **trait**

```
    includes Field(Real, +, *)
    includes TotalOrder(Real)
    includes Natural

    introduces
        2, π :→ Real                                    % constants
        __ − __ : Real, Real → Real                     % ordinary subtraction
        __/__ : Real, Real → Real                       % ordinary division
        __^__ : Real, Nat → Real                        % Nat exponentiation
        root : Nat, Real → Real                         % Nat roots
        cosine : Real → Real

        between : Real, Real, Real → Bool               % is middle arg between others?
        cutoff : Real, Real, Real → Real                % force the middle arg between
        approx : Real, Real, Real → Bool                % first two args approx equal

        natToReal : Nat → Real                          % the usual conversion
        realToNat : Real → Nat                          % roundoff conversion

    asserts
        equations
```

$0 : \mathit{Real} < 1 : \mathit{Real}$
$2 : \mathit{Real} == 1 + 1$

$\forall\ p, q, r : \mathit{Real}$
  $p < q == (p + r) < (q + r)$
  $p - q == p + (-q)$
  $q \neq 0 \Rightarrow (p/q) * q = p$
  $q \neq 0 \Rightarrow (p * q)/q = p$

$\forall\ p : \mathit{Real}, n : \mathit{Nat}$
  $p\hat{}0 == 1$
  $p\hat{}\mathit{succ}(n) == p * (p\hat{}n)$
  $p \geq 0 \Rightarrow \mathit{root}(n, p)\hat{}n = p$
  $p \geq 0 \Rightarrow \mathit{root}(n, p\hat{}n) = p$

$\forall\ r, \mathit{lo}, \mathit{hi} : \mathit{Real}$
  $\mathit{between}(\mathit{lo}, r, \mathit{hi}) == \mathit{lo} \leq r \wedge r \leq \mathit{hi}$
  $\mathit{lo} \leq \mathit{hi} \Rightarrow$
  $\mathit{cutoff}(\mathit{lo}, r, \mathit{hi}) =$
  ( **if** $r < \mathit{lo}$ **then** $\mathit{lo}$ **else if** $r > \mathit{hi}$ **then** $\mathit{hi}$ **else** $r$)

$\forall\ p, q, \mathit{precision} : \mathit{Real}$
  $\mathit{approx}(p, q, \mathit{precision}) ==$
    $\mathit{between}(q - \mathit{precision}, p, q + \mathit{precision})$

$\forall\ n : \mathit{Nat}, r : \mathit{Real}$
  $\mathit{natToReal}(0) == 0$
  $\mathit{natToReal}(\mathit{succ}(n)) == 1 + \mathit{natToReal}(n)$
  $\mathit{realToNat}(r) = n ==$
    $(\mathit{natToReal}(n) - (1/2)) \leq r \wedge$
    $r < (\mathit{natToReal}(n) + (1/2))$

**implies**
  $\forall\ r, \mathit{lo}, \mathit{hi} : \mathit{Real}$
    $\mathit{lo} \leq \mathit{hi} \Rightarrow \mathit{between}(\mathit{lo}, \mathit{cutoff}(\mathit{lo}, r, \mathit{hi}), \mathit{hi})$
  **converts** $- : \mathit{Real}, \mathit{Real} \rightarrow \mathit{Real}$,
    $\hat{}\ : \mathit{Real}, \mathit{Nat} \rightarrow \mathit{Real}$,
    $\mathit{between} : \mathit{Real}, \mathit{Real}, \mathit{Real} \rightarrow \mathit{Bool}$,
    $\mathit{natToReal}$

## A.7.9 Natural Numbers

% Nat is the sort of natural numbers, 0,1,....
% This trait includes the theory of Cardinals, but introduces additional
% properties and names for a few specific numbers.

*Natural* : **trait**

**includes** *Cardinal*(*Nat* **for** *Card*, __^__ **for** __**__)

**introduces**

$2, 3, 4, 5, 6, 7, 8, 10, 14, 15, 16, 256 : \rightarrow Nat$
$between : Nat, Nat, Nat \rightarrow Bool$
$cutoff : Nat, Nat, Nat \rightarrow Nat$
$mod : Nat, Nat \rightarrow Nat$

**asserts**

**equations**

$2 == succ(1)$
$3 == succ(2)$
$4 == succ(3)$
$5 == succ(4)$
$6 == succ(5)$
$7 == succ(6)$
$8 == succ(7)$
$10 == 8 + 2$
$14 == 10 + 4$
$15 == succ(14)$
$16 == succ(15)$
$256 == 16 * 16$

$\forall\ lo, n, hi : Nat$
$between(lo, n, hi) == lo \leq n \land n \leq hi$
$lo \leq hi \Rightarrow$
$cutoff(lo, n, hi) =$
( **if** $n < lo$ **then** $lo$ **else if** $n > hi$ **then** $hi$ **else** $n$)

**implies**

$AbelianSemigroup(Nat, +)$,
$AbelianSemigroup(Nat, *)$,
$Distributive(Nat$ **for** $T)$

# Appendix B

# Critique of the Informal GCS Requirements

NASA's Guidance and Control Software (GCS) experiment is a study of software development methods. As part of this experiment, Research Triangle Institute (RTI) wrote an informal software requirements document for GCS. This document is titled "Software Requirements: Guidance and Control Software Development Specification", and was released in June 1990 [7]. We will refer to this document as "SR" throughout this chapter.

ORA Corporation has written a formal specification for GCS, based on the information in SR. This formal specification appears in Appendix A.

In the process of writing the formal specification we noted some problems, both potential and actual, with the SR specification. This document lists the problems noted. Most of the details in SR are in its descriptions of the functionality of each GCS module. We devote one section here to each module in which problems were noted. The first section, though, discusses a few problems found in the overall GCS requirements.

## B.1 General Requirements

Several problems arise in SR's general requirements for GCS. Of these, the most important are the following:

- SR provides too much detail when stating the GCS requirements on functionality and timing. In particular, it describes intermediate variables to be used in the code, a decomposition of the functionality into modules, and gives not only an overall timing requirement but timing constraints for each module and a schedule for activating modules. This detail makes SR essentially a software design, rather than merely an expression of requirements.

- SR provides too little detail about the requirements on precision of computations.

In section 3.3 we place these problems within the context of a larger discussion of requirements specification for control software. We do not discuss them further here.

More specific problems follow:

- Page 12's description of the GCS control needs to be updated: according to the specification for AECLP, proportional-integral-derivative (PID) control is used for pitch and yaw.

- The data flow diagram on page 26 does not reflect all the flows actually required by the rest of the specification. For example, ATMOSPHERIC_TEMP and AR_ALTITUDE are both part of the SENSOR_OUTPUT data, according to the data dictionary. But both serve as inputs to at least one of the sensor processing modules. Inputs to sensor processing other than from the sensors themselves are not shown in the data flow diagram.

## B.2 AECLP

SR's requirements for Axial Engine Control Law Processing (AECLP) have these problems:

- Table 5.2 defines pitch and yaw using different conventions: if pitch in this table is right-handed, then yaw is left-handed, or vice-versa. This discrepancy is not important if the PID coefficients are chosen with the correct sign, but the signs of these coefficients are also questionable (see next item).

- There are nine PID coefficients in Table 5.2. The data dictionary requires GAX to be positive, but allows the other eight to have either sign. However, SR explicitly negates another of these, GR, thus suggesting that the coefficients will be chosen positive.

  One would expect that the three coefficients in each of the PID control laws would have the same sign. The P and I terms are restoring forces, proportional to the error and its integral, respectively. The D term damps oscillations, and should therefore add to the P and I terms when the motion is increasing the error.

  Therefore the negative sign modifying the GR term is either wrong or superfluous.

- In the control law for thrust, the purpose of the GAX term, not just its sign, is questionable. As written, the term tends to be de-stabilizing: the greater the downward acceleration, the faster the downward thrust decreases, leading to greater downward acceleration. (This conclusion depends on the fact that the data dictionary requires GA to be positive.) Therefore successful control must depend on a choice of coefficients that makes the GVE and GVEI terms more important than the GAX term in correcting deviations from the intended flight path.

  A term such as the GAX term is still needed, though, to determine the magnitude of thrust in the absence of deviations from the intended flight path. Why should this term depend on the measured acceleration in the lander's frame of reference? Would it not be better to replace the GAX term with a function of altitude and attitude approximating the thrust needed, e.g., a term proportional to

  $$gravity/cos\theta + dx_v/dt \ d(contour)/d(altitude)$$

  where $\theta$ is the angle between the verticals in the lander's and planet's coordinates, and *contour* is the precomputed velocity-altitude contour?

- The data dictionary's description of OMEGA, "gain of angular velocity", is wrong. OMEGA's intuitive physical significance is the reciprocal of the time scale over which changes in thrust are integrated.

- The matrix equation on page 36 that determines INTERNAL_CMD is inconsistent with the diagram of the lander on page 9. SR does not say how the three axial engines are numbered, but it does not matter: pitch control, according to the equation, uses all three engines, but according to the diagram it depends on only two; yaw control, according to the equation, uses two engines only, but according to the diagram it must use all three. Obviously pitch and yaw are interchanged here.

- The equation for INTERNAL_CMD also suggests a problem with signs. As written, GP1 and GP2 must have opposite signs so that they supply torque but not net thrust. The data dictionary does not rule out this possibility, but the specifier could have chosen, based on the picture on page 9, to make all the constants in this equation positive.

- AE_STATUS should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy".

## B.3 ARSP

SR's requirements for Altitude Radar Sensor Processing (ARSP) are unambiguous, but the assumptions that seem to underlie those requirements are questionable.

If AR does not receive a radar echo in the current frame, it fits a cubic polynomial to the previous four altitude values. Two questions arise:

- Why are four values used? Using more values increases the probability of using some that are unreliable (i.e., themselves based on extrapolation). Using fewer values may decrease the precision of the extrapolation. Using fewer values simplifies fitting, and therefore makes real-time deadlines easier to meet. So there is a trade-off, and perhaps four values are optimal. SR does not provide enough information to decide.

- Why is a cubic fit to the four values? Using a cubic suggests that the first three derivatives of altitude are non-negligible over four frames. But in acceleration processing (ASP), the third derivative is assumed to be negligible in extrapolating acceleration values. ASP's extrapolation is sometimes used in guidance processing to get new values of velocity and altitude, so precision is no less important in ASP than ARSP.

  Therefore fitting a quadratic to the altitude values is probably better than fitting a cubic.

  If the first and second derivatives are non-negligible, then why not use the values for these from guidance processing (GP) to constrain the ARSP fit, rather than letting the fit determine them? The result of this constrained fit, an altitude, is likely to be more precise than the guidance processing value for altitude, because the constrained fit uses several recent direct measurements of altitude rather than the single most recent value used in GP.

  Whether to make better use of available data to get a more precise value for altitude is a decision that depends on GCS's specific requirements on precision. These requirements are not stated in SR.

## B.4 ASP

SR's requirements for Accelerometer Sensor Processing (ASP) have the following problems:

- The requirement for converting raw counter data to an acceleration value seems to contradict itself. First SR says: "The sign of the counter will always be positive, but the offset given in **A_BIAS** will be negative or zero, so if the magnitude in **A_COUNTER** is smaller than that of **A_BIAS**, the acceleration is negative." To reinforce this, SR says: "Each accelerometer has a characteristic DC bias (**A_BIAS**) which must be removed from the signal prior to conversion." Both statements suggest a conversion of the form

$$acceleration = gain * (A\_COUNTER + A\_BIAS)$$

SR also says, however: "The acceleration is a linear function of its **A_COUNTER** value where the gain specifies the slope and the offset (**A_BIAS**) specifies the intercept." To reinforce this, SR writes an equation for converting each acceleration component:

$$acceleration = A\_BIAS + gain * A\_COUNTER$$

Which is right? The GCS formal specification uses the second interpretation.

- The misalignment correction is wrong. SR defines an **ALPHA_MATRIX** of small correction angles, and defines the small angles in the adjacent text. The matrix shown, however, must be transposed and the alphas negated in order to follow the text.

- The sample standard deviation is defined incorrectly. The definition is the standard deviation of the parent distribution, which will differ from the sample standard deviation unless the mean is known with perfect accuracy (which is not the case here).

- The algorithm for eliminating flaky acceleration values is questionable. Its intent seems to be that an acceleration value is unreliable ("unhealthy") if it is too far from the mean of recent values. However, if the mean itself depends on unreliable values, then the algorithm forces the current value to be reliable. If we suppose the reliability of each measurement to be independent of the others, this algorithm is absurd: the current value becomes more reliable simply because its predecessors are less so.

  A better approach almost certainly exists. Why is a mean of recent measurements used instead of a more complicated extrapolation? Presumably because the change in acceleration over four frames is negligible. Why is a mean taken for just three measurements? Presumably because the probability of flaky values is small enough that getting two of them in four consecutive frames is unlikely. So why not take the mean of the three most recent *reliable* values? This will almost never need more than one additional previous frame, and the change in acceleration over *five* frames is probably still negligible.

  Justifying an alternate approach, of course, demands an understanding of the detailed dynamics and of the sensor precision. These details are not available in SR.

## B.5 CP

SR's requirements on Communications Processing (CP) have these problems:

- On page 46, SR notes that the diagonal terms of G_ROTATION should not be sent. G_ROTATION is a history of vectors, so there are no diagonal terms to send. Probably the reference should be to GP_ROTATION, whose diagonal terms are always zero. However, GP_ROTATION should not be sent at all, because it is essentially an intermediate variable determined entirely by G_ROTATION.

  This mistake, though minor, is but the tip of an iceberg. Generally, SR should not be listing the variables to send because specific decision about GCS's data structures should be at the discretion of the programmer. Some of the "variables" listed in CP, such as GP_ROTATION, might not even exist in the implementation. This general problem of overspecification is discussed in chapter 3

- C_STATUS should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy". And even if C_STATUS did depend on whether CP was working OK, there would be no reason to transmit it as part of a communication packet: the arrival of uncorrupted packets should be the best indication of CP's health.

## B.6 GSP

SR's requirement on Gyroscope Sensor Processing (GSP) seems to contradict itself. The raw sensor data for each axis is a counter that holds both a sign bit and a 14 bit magnitude. The text describes the conversion: "The rotation rate is linear with respect to the unprocessed gyroscope values, i.e., the lower 14 bits must be converted." Therefore, when the sense of the rotation is included, the conversion should be written

$$rotation = sign(counter) * (offset + gain * magnitude(counter))$$

But SR goes on to write the conversion as

$$rotation = offset + gain * counter$$

Which is right?

The GCS formal specification uses the former interpretation.

G_STATUS should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy".

## B.7 GP

SR's requirements for Guidance Processing (GP) have these problems:

- The equation on page 54 for the derivative of dynamical quantities is wrong. To see this, note that it is dimensionally incorrect in the case of velocity: the left hand side has dimensions of an acceleration, but the right hand side correction term has dimensions of a velocity. The problem could be fixed by using a difference equation instead, e.g.,

  $$\Delta(variable) = \alpha \times variable\Delta t + \beta\Delta t + correction\ term$$

  However, this fix effectively requires Euler integration, while SR explicitly allows better integration algorithms to be used in this case.

  A better fix would be to eliminate the correction terms and replace them with conditionals in the specification logic (see next item).

- The terms with the `K_ALT` and `K_MATRIX` factors are misleadingly called "correction terms". Their real purpose is to choose measured values from ARSP and TDLRSP if they are reliable, or choose values gotten by integration in GP otherwise. It would be better to state this requirement explicitly, than to embed it as a kludge in equations for the lander's dynamics.

## B.8 RECLP

`RE_STATUS` should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy".

## B.9 TDLRSP

SR's requirements on Touch Down Landing Radar Sensor Processing (TDLRSP) are unambiguous, but this fact only becomes clear after making an educated guess about the intentions behind the requirements and re-deriving the "requirements" from the intentions. The document would be much clearer if either

- it gave a few more clues about the intentions, or
- the intentions were stated as the requirements, and the software engineer were allowed to derive consequences from them.

The formal requirements specification for GCS follows the latter course. This course has two advantages:

1. The requirements are more abstract, and as a result are more easily adapted if the design of the spacecraft changes.
2. The programmer reading the requirements is not left guessing.

To understand the problem, compare Figure 5.4 on page 69 with Figure 5.3 on page 66. Figure 5.3 shows the directions of the four radar beams, while Figure 5.4 names the direction cosines for the beam in the first octant. The programmer is expected to realize that the directions of the other three beams are also gotten from the same direction cosines. But how? Reflection in the coordinate planes? Rotation through 90 degrees? Other? Only after re-deriving the expressions in Table 5.10 can one be sure that it must be reflection, because only in that case will the values of two beams uniquely determine one component of the velocity.

SR's figure 5.3 does suggest that the beam directions determine a rectangle, centered on the origin, and parallel to the coordinate axes. SR should at least make this explicit.

The intentions behind the requirements can be stated much more succinctly than Table 5.10: use every available beam measurement to determine as many components of the velocity as possible. If a component of the velocity is underdetermined, flag its value as unreliable. If a component of the velocity is overdetermined, average the values gotten from the different sets of beam measurements that determine it.

SR does not make clear why this intention is good enough. If in some frame only two beam measurements are available, exactly one velocity component is determined. The other components are gotten from Guidance Processing (GP) by integrating the acceleration. These GP components are considered less reliable because they are not gotten from direct measurement. But this method wastes the more precise information available in TDLR's direct measurement: two measurements should determine two degrees of freedom, not just one. Why isn't the GP information averaged with the TDLR measurement, weighted by the relative precision of the two? This averaging would yield a more precise value of velocity in this case.

Finally, TDLR_STATUS should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy".

## B.10 TSP

SR's requirements on Temperature Sensor Processing (TSP) have the following problems:

- Paragraph 1 is misleading. The requirements on TSP are intended to maximize the *precision* of measurements; the effect on the accuracy depends on random factors in each measurement. The thermocouple pair sensor is more precise in the range for which its calibration gives accurate results; the solid-state sensor gives accurate results over a wider range.

- The requirements on the calibration of the thermocouple pair appear incomplete. The thermocouple pair is calibrated by a parabola near the ends of its usable range. SR states requirements that determine these parabolas. One of the requirements is "The upper (and lower) parabolas are defined so that the temperature goes up (or down) as the square of the measurement value." Evidently $(T - T_0) = \alpha(M - M_0)^2$, where $T_0$ and $M_0$ are to be determined by the two other requirements given in SR. Are we to assume that $\alpha = \pm 1 degree\ Centigrade/counts^2$?

  This is the only instance in SR where a dimensional constant used in sensor calibration is not specified in the data dictionary, and not loaded at GCS initialization. Perhaps neglecting $\alpha$ is a side-effect of simplifying the Viking lander software to become GCS.

  This neglect may not turn into a problem, because it is unlikely the dimensions chosen for temperature will be changed during the design process. However, the specification of TSP is not invariant under such changes.

- The requirement on when to use the thermocouple pair is certainly wrong. SR states "If the temperature derived from SS_TEMP falls within the accurate temperature response zone of the thermocouple pair, ..., then the value returned by the thermocouple pair should be used;..." This suggests the solid-state sensor temperature must be calculated first, then a decision made based on the result of the measurement. There are two reasons not to do this:

  1. SR specifies elsewhere that the thermocouple pair is calibrated accurately over a certain range of raw sensor readings, not over a range of temperature.
  2. Even if we use the thermocouple's calibration to convert the range of sensor readings into a range of temperatures, it is still better to decide which sensor to use based on the raw readings. Every measurement is imprecise, especially those of the solid-state sensor. Therefore, it is possible for the raw sensor readings to fall outside the thermocouple pair's range of accuracy, but for the solid-state sensor to convert the raw readings into a temperature value apparently inside the range. Converting the thermocouple pair's reading could then yield an inaccurate result. The likelihood of this occurring depends on the imprecision of the solid-state sensor.

- TS_STATUS should be deleted from the specification. There are no stated conditions under which it is given a value other than "healthy".

## B.11 Data Dictionary

There are several trivial mistakes in the Data Dictionary which should be fixed if a new requirements document is produced.

- `AE_TEMP` has three values, so its data type cannot be logical*1.
- The dimension *radians* is used in various places in the data dictionary, including for `TDLR_ANGLES`, so it should also be used for the angles in `ALPHA_MATRIX`.
- How could `FRAME_COUNTER` be `EXTERNAL`?
- The description of `G_OFFSET` uses the name `ROTATION_RAW`, which does not appear elsewhere in the dictionary.
- `RE_SWITCH` is also used in RECLP.

# Bibliography

[1] Philip R. Bevington. *Data Reduction and Error Analysis for the Physical Sciences.* McGraw-Hill. 1969.

[2] Robert S. Boyer, Milton W. Green, and J Strother Moore. The use of a formal simulator to verify a simple real time control program. In *Beauty is Our Business.* Springer-Verlag, 1990.

[3] Software considerations in airborne systems and equipment certification (DO-178-B.5). RTCA paper no. 591-91/SC167-164. November 1991. 1140 Connecticut Avenue. NW. Suite 1020. Washington. DC 20036.

[4] Janet R. Dunham. Experiments in software reliability: Life-critical applications. *IEEE Transactions on Software Engineering; Special Issue on Software Reliability,* January 1986.

[5] David Guaspari. Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering,* 16:1058–1075, September 1990.

[6] John V. Guttag. James J. Horning. and Andres Modet. Report on the larch shared language: Version 2.3. Technical report. Digital Equipment Corp. Systems Research Center. April 1990.

[7] B. Edward Withers et al. Software requirements: Guidance and control software development specification. Technical report, Research Triangle Institute, June 1990.

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE April 5, 1994 | 3. REPORT TYPE AND DATES COVERED Contractor Report |

| | |
|---|---|
| 4. TITLE AND SUBTITLE Using Formal Specification in the Guidance and Control Software (GCS) Experiment | 5. FUNDING NUMBERS C NAS1-18972 WU 505-64-10-51 |
| 6. AUTHOR(S) Doug Weber and Damir Jamsek | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates 301 Dates Drive Ithaca, NY 14850-1326 | 8. PERFORMING ORGANIZATION REPORT NUMBER TM-92-0046 |
| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23681-0001 | 10. SPONSORING / MONITORING AGENCY REPORT NUMBER NASA CR-194884 |

11. SUPPLEMENTARY NOTES
Task 7 Report
Langley Technical Monitor: Sally C. Johnson

| | |
|---|---|
| 12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 60 | 12b. DISTRIBUTION CODE |

13. ABSTRACT *(Maximum 200 words)*

The goal of this task was to investigate how formal methods could be incorporated into a software engineering process for flight-control systems under DO-178B and to demonstrate that process by developing a formal specification for NASA's Guidance and Controls Software (GCS) Experiment. GCS is software to control the descent of a spacecraft onto a planet's surface. The GCS example is simplified from a real example spacecraft, but exhibits the characteristics of realistic spacecraft control software. The formal specification is written in Larch.

| | |
|---|---|
| 14. SUBJECT TERMS Formal Methods, Specification, Guidance and Controls | 15. NUMBER OF PAGES 108 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |